

DEITEL 编程金典

Python 编程金典

[美] H. M. Deitel, P. J. Deitel 著
J. P. Liperi, B. A. Wiedermann 译
周 靖

清华大学出版社
北 京

内 容 简 介

本书由全球著名的编程培训专家 H. M. Deitel 博士领头编写, 解释了如何将 Python 用做常规用途, 编写基于 Internet 和 Web 的、数据密集型的、多层的客户端/服务器系统。书中采用作者独创的“活代码”教学方法, 揭示了 Python 这一语言的强大功能。与此同时, 本书还提供了大量的示例代码和编程技巧与提示, 帮助读者搭建良好的知识结构, 养成良好的编程习惯, 指导读者如何避免常见的编程错误以及写出高效、可靠的应用程序。

本书适合对 Python 感兴趣的读者阅读和参考。

EISBN: 0-13-092361-3

Python How To Program

H. M. Deitel, P. J. Deitel, J. P. Liperi, B. A. Wiedermann

Copyright © 2002 by Prentice-Hall, Inc.

Original English language edition published by Prentice-Hall, Inc.

All right reserved.

For sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macau SAR).

仅限于中华人民共和国境内（不包括中国香港、澳门特别行政区和中国台湾地区）销售发行。

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签, 无标签者不得销售。

北京市版权局著作权合同登记号: 图字 01-2002-4423 号

图书在版编目 (CIP) 数据

Python 编程金典/ (美) 迪特尔等著; 周靖译.—北京: 清华大学出版社, 2003

(DEITEL 编程金典)

书名原文: Python How To Program

ISBN 7-302-06642-6

I. P... II. ①迪... ②周... III. 软件工具 程序设计 IV. TP311.56

中国版本图书馆 CIP 数据核字 (2003) 第 036550 号

出 版 者: 清华大学出版社 (北京清华大学学研大厦, 邮编 100084)

<http://www.tup.com.cn>

<http://www.tup.tsinghua.edu.cn>

责任编辑: 文开棋

印 刷 者: 清华大学印刷厂

发 行 者: 新华书店总店北京发行所

开 本: 787 × 1092 1/16 印张: 37.25 字数: 1270 千字

版 次: 2003 年 6 月第 1 版 2003 年 6 月第 1 次印刷

书 号: ISBN 7-302-06642-6/TP · 4970

印 数: 0001 ~ 2000

定 价: 88.00 元

前言

欢迎进入 Python 编程世界！Python 是一种强大的常规用途程序语言，尤其适合开发基于 Internet 和 Web 的、数据库密集型的、多层的客户机/服务器系统。本书讲解了大量先进的计算技术，它是我们的第二本有关开放源码程序语言的参考书。^①

在我们编写本书时，Python 2.2 刚刚发布。为此，我们进行了艰苦的工作，以便将 Python 2.2 的功能合并到本书中。附录 B 将讲解 Python 2.2 的其他一些特性。

希望读者通过本书获取有用的信息，既能感到适度的挑战，又能从中获得无穷乐趣！本书写作过程令人愉快。Deitel & Associates 的团队长期致力于程序语言教科书和 e-Learning 素材的开发。我们涉足几乎每一种主流程序语言。在编写本书过程中，我们也注意到一些特别的地方。我们的开发者和撰稿人对 Python 给与高度评价。它强大的功能、高度的可靠性和编码的简洁性，给人留下深刻印象。他们喜欢 Python 能让他们运用自如。他们喜欢这种开发源码的软件开发世界，为 Python 而开发的模块正在与日俱增。

无论老师、学生、有经验的专业程序员还是新手，都能通过本书汲取有益的知识。Python 是一种出色的程序语言，也是开发具有工业强度的商业应用程序的优秀语言。对于学生和新手级程序员，通过前几章的学习，可打下良好的基础。我们讨论了许多程序开发模型，包括结构化编程、基于对象的编程、面向对象的编程以及事件驱动的编程等。对于专业开发人员，我们则选用 Python 真正强大的功能来创建实用的、进行了完全实现的系统。这部分的重点在于第 23 章的案例分析，它详细讲解了如何构建一个真正的网上书店。

本书涉及了所有标准主题，包括数据类型、运算符、控制结构、算术运算、字符串、决策、算法开发、函数和随机数/模拟等等。

本书的特色之一是全面讲解了数据结构，书中首先介绍了 Python 的内建结构——列表、元组和字典。之后，还对包括队列、堆栈、链表和二叉树在内的传统数据结构进行了深入讲解。

本书强调了 Internet 和 Web 开发——我们首先介绍了 CGI，并在随后几章用它来构建基于 Web 的应用程序。我们用整章篇幅（第 25 章）介绍了 PSP（Python Server Pages，Python 服务器页），并利用它改编了第 16 章介绍的一个论坛案例。

本书用 3 章篇幅详细介绍了面向对象编程，涉及的主题包括类、封装、对象、属性、方法、构造函数、析构函数、自定义、运算符重载、继承、基类、派生类和多态性等等。

本书透彻讲解了如何用 Tkinter 进行 GUI（Graphical User Interface，图形用户界面）编程，涉及的主题包括事件驱动编程、标签、按钮、复选框、单选钮、鼠标事件处理、键盘事件处理、布局管理器以及一系列高级 GUI 功能，利用它们可创建和处理菜单和滚动组件。

我们讨论了如何利用异常处理使程序更“健壮”。Python 强大的字符串处理功能在此得到了深入讲解。至于正则表达式的主题，虽然它不易于理解，但由于它功能强大，所以我们也进行了详尽的解释。

我们讨论了文件处理、顺序访问文件、随机访问文件（以及 shelve 模块），同时还开发了一个事务处理程序，论述了对象序列化的问题。通过讨论文件处理，为后来的 Python 数据库编程奠定了良好的基础，后者通过 Python 的 DB-API（Database Application Programming Interface，数据库应用程序编程接口）来实现。我们讨论了关系数据库模型，并概述了 SQL（Structured Query Language，结构化查询语言）。

许多人都熟悉 HTML，但很少有人知道万维网协会（W3C）——HTML 技术的创建者——已声称 HTML 已成为“过去”，不会继续开发它。全世界正逐步过渡到 XML（eXtensible Markup Language，可扩展标记语言）。在这期间，Web 开发将采用一种名为 XHTML 的过渡技术。本书许多应用程序都将采用 XHTML。至于 XML 的常规主题，将采用一整章（即第 15 章）的篇幅来介绍它。对如今的 Web 应用

^①第一本是《Perl 编程金典》，清华大学出版社 2002 年出版，ISBN 7-302-05751-6。

程序开发者来说，这是必须掌握的。然后，我们将另起一章（第 16 章），专门讲解 Python 的 XML 处理技术，并提供一个详细的案例分析，运用 CGI 和 XML 来构建论坛。

计算机应用程序通常能很好地一次做一件事。今天，较高级的应用程序需要同时做许多事，在计算机领域内，我们更喜欢将其称为“并发性”。我们会分别用整章的篇幅来讲解进程管理（第 18 章）和多线程处理（第 19 章）。Python 程序员利用这些技术，可做到以前只有系统程序员在操作系统的级别上才能做到的事情。

我们讨论了联网问题，包括 Web 使用的 HTTP 协议，使用流套接字进行的客户机/服务器联网，使用数据文报进行的无连接的客户机/服务器交互，另外还使用多线程服务器，实现了一个客户机/服务器的 Tic-Tac-Toe（即三连棋）游戏。

我们全面讨论了常见的计算机安全问题，并讲解了 Python 特有的一些安全问题。讨论了如何使用模块 Bastion 在一个限制环境中执行恶意代码。另外，还演示了如何用模块 rotor 对文本进行加密。

作为本书的一个重点，第 23 章展示了一个详尽的案例分析，它使用前几章和本书附录讨论的许多技术来实现一个电子商务网上书店。我们介绍了 HTTP 会话和会话跟踪技术，并将这个书店构建成为一个多层的客户机/服务器系统，它有能力处理大批量的客户，其中包括标准 Web 浏览器（使用 XHTML）和无线客户（使用 WML 和 XHTML Basic）。

我们用整章（第 24 章）的篇幅讲解多媒体所涉及的主题。使用 3D 图形例子介绍了 PyOpenGL，并介绍了 3D 环境 Alice，它提供的对象可通过 Python 脚本“动”起来。我们通过设计一个 CD 播放器，一个影片播放器和一个太空船游戏，演示了 pygame。

认识到服务器端开发的重要性之后，我们将展示 PSP（Python 服务器页），它可取代 CGI。另外，我们将论坛案例从 CGI 技术转换成了 PSP。

本书提供了两个附录，均与 Python 有关。其中，附录 A 介绍了 Python 开发环境，附录 B 介绍了 Python 2.2 的其他特点，其中还讨论了迭代器、生成器和嵌套作用域。

阅读本书的过程中，不管遇到什么问题，都请联系 deitel@deitel.com，来信必复。另外，请经常访问我们的网站 www.deitel.com，并订阅免费的 The Deitel BUZZ 电子刊物。我们会通过网站及电子刊物介绍最新的 Python 信息以及我们推出的其他产品和服务。

本书特色

本书具有许多特色，包括：

- **代码清洗（code washing）**。这是我们自己创造的一个术语，是指对程序进行全面格式化，为其精心添加注释，并使其具有一个开放布局。程序代码组合成小的、结构清晰的块，这大大增强了可读性——这正是我们要达到的一个重要目标，尤其是全书总共含有 14 930 行代码！
- **面向对象编程**。面向对象编程是目前广泛采用的一种编程技术，用于开发健壮的、可重用的软件。Python 被设计成一种面向对象语言，本书全面讨论了 Python 的各种面向对象特性。数据完整性是 Python 中尤其要关注的一个问题。所有 Python 类数据默认都是公共的，但几种技术可用于确保数据完整性。我们用 3 章篇幅详细讨论了这些问题以及其他面向对象的主题。其中，第 7 章介绍如何创建类，并讨论了公共、私有和 get/set 方法。第 8 章解释如何使创建的类具有自定义行为，比如运算符重载、字符串表示、列表和字典行为以及用于自定义属性访问的方法等。第 9 章对这些概念进行了进一步扩展，我们讨论了如何创建新类，令其“吸收”现有类的功能。通过这一章的学习，读者会熟悉一些关键概念，比如多态性、抽象类和具体类等，利用它们可更方便地处理一个继承层次结构中的对象。本章最后讨论了 Python 2.2 提供的其他面向对象能力，其中包括“属性”（Properties）。
- **数据库应用程序编程接口**。数据库存储着大量信息，个人和单位需要访问它们以处理各种事务。数据库管理系统（DBMS）供单位和个人用来操纵数据库——Python 提供了相应的数据

库应用程序编程接口 (DB-API)，以访问数据库管理系统。第 17 章详细介绍了这些功能，另外还介绍了用于查询 MySQL 数据库的结构化查询语言 (SQL)。

- **XML。**可扩展标记语言 (XML) 在软件开发领域和电子商务社区获得了蓬勃发展。由于 XML 是一种与平台无关的技术，用于描述数据和创建标记语言，所以 XML 的数据移植性能与 Python 的可移植应用程序和服务较好地集成在一起。第 15 章介绍了 XML，我们讨论了基本的 XML 标记和技术，比如 DTD 和 Schema，它们用于校验 XML 文档内容。第 16 章则解释了如何用文档对象模型 (Document Object Model, DOM) 来处理 XML 文档，以及如何通过可扩展样式表语言转换 (eXtensible Stylesheet Language Transformation, XSLT)，将 XML 文档转换成其他文档类型。本章还介绍了 DOM 的一种替代物，名为 Simple API for XML (简称 SAX)，它充当用于 XML 的一个基于事件的 API。
- **公共网关接口 (CGI) 和 Python 服务器页 (PSP)。**因特网和万维网已深入人们的日常生活，交互式网站是商业成功的关键。第 6 章和第 25 章介绍了服务器端 Web 技术，开发者利用它们可创建交互式的、基于 Web 的应用程序。第 23 章提供了一个详细的案例分析，它综合运用 MySQL、XML、XHTML、XHTML Basic、层叠样式表 (CSS)、XSLT、CGI 和无线标记语言 (Wireless Markup Language, WML) 来构建一个动态电子商务应用程序。本书演示了一个 XML 论坛的两种实现方式，用户可将自己的文章张贴到在线论坛。第 16 章使用的是 CGI，第 25 章使用的则是 PSP。
- **图形用户界面 (GUI)。**Python 没有内建图形用户界面功能，但有许多模块可供使用，它们提供对现有的 GUI 软件的访问途径。第 10 章和第 11 章讨论了 Tkinter 模块 (包括在 Python 标准库中)，它允许 Python 程序员访问 Tool Command Language/Tool Kit (Tcl/Tk) 这一流行的 GUI 工具包。利用这些编程工具，开发者可快速、方便地创建图形程序。利用在这几章所学的知识，读者可为本书其余部分的程序开发 GUI。第 11 章还讨论了模块 Pmw，它利用 Tkinter 提供更复杂的 GUI 组件。
- **多媒体。**多媒体功能可生成具有丰富视听感受的强大应用程序，由此增强用户的体验。可利用几个 Python 模块创建令人印象深刻的多媒体应用程序。第 24 章探讨了 PyOpenGL 和 Alice 的功能，它们可创建 3D 图形，并让它“动”起来。同时还讨论了 pygame，它所包含的模块便于开发者访问强大的多媒体库。第 24 章使用 pygame 创建一个 CD 播放器、一个电脑游戏以及一个影片播放器。
- **多线程处理和进程管理。**计算机可并发执行大量任务，比如同时打印文档、从网络下载文件和在 Web 上冲浪等等。利用多线程处理技术，应用程序可执行并发性任务。Python 的多线程处理和进程管理功能尤其适合今天高度复杂的、多媒体密集型的、数据库密集型的、基于网络的、基于多处理器的以及分布式的应用程序。第 18 章讨论了并发性与进程间通信；第 19 章详细讨论了多线程处理的问题，其中详细解释了 Python 的 Global Interpreter Lock (即全局解释器锁，负责管理线程执行)。本章还通过几个例子，介绍了常见的线程同步机制。
- **文件处理和序列化。**大多数应用程序都要在磁盘上读写数据。Python 针对数据存储和获取提供了几项高级功能。第 14 章讨论了用于存储顺序数据的基本文件对象、用于存储随机访问数据的 shelve 对象，以及用于将整个对象序列化到磁盘的 cPickle 模块。

此外，本书还讨论了其他许多主题。要详细了解每章的特点，请参阅 1.6 节。

Python 2.2 的特性

本书出版时，喜闻 Python 2.2 正式版刚刚发布。然而，本书所有示范代码都通过了 Python 2.2b2 (即 Beta 2) 和 Release Candidate 1 的测试。测试平台包括 Windows 和 Linux 操作系统。我们在各章尽可能介

绍 Python 2.2 的特性和功能。[†]本书要介绍 Python 2.2 的以下特性。

Floor 除法和 True 除法：Python 2.2 引入新运算符 (`//`) 进行 Floor (整数) 除法。在此之前的 Python 版本中，除法运算符 (`/`) 的默认行为是 Floor 除法；在 2.2 以后的版本，默认行为变成 True (浮点) 除法。通过定义两个除法运算符，Python 新版本可在同时使用了整数及浮点除法的程序中，避免出现类型混淆的问题。本书 2.6 节讨论了这两种除法的区别，并解释了程序如何更改除法运算符 (`/`) 的默认行为，令其执行 True 除法。

嵌套作用域：Python 2.2 引入了嵌套作用域的概念，它意味着嵌套的类、方法和函数现在可访问其封闭作用域中定义的变量。这种行为尤其适合编写 lambda 表达式。第 4 章讨论了 Python 的基本作用域规则，并提供了一系列万维网资源，便于读者更深入地了解嵌套作用域。如程序员在一种功能性编程模型中使用 Python，作用域的嵌套就显得非常重要。附录 B 更详细地讨论了嵌套作用域。由于本书强调的主要是面向对象的编程风格，所以只指出了使用嵌套作用域的一个高级动机，并推荐了可进一步参考的资源，便于读者在需要时了解 Python 中的嵌套作用域和功能性编程。

更多的面向对象功能：Python 2.2 的大多数新特性是为语言添加更多的面向对象功能。第 8 章和第 9 章介绍了其中的一些新特性。第 8 章讨论了如何重载一个由程序员定义的类，以便为运算符 (包括新运算符 `//`) 定义行为。介绍了一个字典方法，它也是 Python 2.2 版本新增的，便于程序用 `if/in` 语句检测字典中是否包含一个特定的键。第 9 章讨论开发者们期待已久的新特性——允许从内建类型继承程序员定义的类。本章展示了一个实例，它继承自内建类型 `list`，目的是实现由程序员定义一个列表，其中只包含惟一性的元素。还讨论了其他面向对象特性，其中包括静态方法，`__slots__` (用于定义类中可能包含的一个对象的属性)，方法 `__getattr__` (客户每次访问一个对象的属性时执行)，以及属性 (允许类定义 `get/set` 方法，以便在客户访问一个属性时执行)。

迭代器：附录 B 介绍的其他 Python 2.2 特点中，有的未在正文中详细讲解。附录 B 首先全面地探讨了迭代器——用于遍历一系列值的特殊对象。B.2 节提供了两个例子，它们展示了由程序员定义的迭代器类，并演示该类的一个客户如何使用迭代器从一个序列中获取值。第一个例子展示了如何定义一个类，使它的对象支持迭代器；第二个例子展示了一个计算机猜谜游戏，它展示如何利用迭代器处理长度不确定的序列。Python 2.2 采用了新的迭代器机制后，性能比以前的版本有了显著改进。另外，软件设计也因为程序员能分离迭代行为和随机访问行为而得以改进。

生成器：这是一种“可恢复函数”，能记住两次调用期间的状态。生成器也有利于性能和设计的改进。通常，程序中可以写一个生成器，采用一种简单、直观方式定义如何生成一个序列的元素。生成器还有利于执行重复性任务，或要求复杂逻辑和状态信息才能完成的任务。B.2 节围绕上述问题讨论了生成器，并定义了两个版本的生成器来计算斐波拉契序列。第一个版本不确定地生成序列中的下一个值；第二个生成所有序列值，直到包括用户自定义的第 n 个值。

万维网访问

本书 (以及我们的其他出版物) 所有示例代码都可从以下网站下载：

www.deitel.com

www.prenhall.com/deitel

注册过程非常简单。建议下载所有例子，并在阅读时运行相应的程序。修改例子，可马上看到修改效果——这是提升编程水平的有效方式之一。上述网站还解释了如何安装本书用到的各种软件 (比如 Apache Web Server)。网站还提供其他 Web 服务器和软件的安装指南 (注意，它们是有版权的。学习时可任意使用，但未经 Prentice Hall 和作者的书面许可，不得采取任何方式重新出版它的任何部分)。

[†] 阅读本书前，先从 python.org 下载最新 Python 版本。新版本发布后，我们会对本书代码进行测试，并在 www.deitel.com 进行相应的更新。阅读每一章之前，最好能访问我们的网站查看这些更新。

目 录

第 1 章 绪论	1
1.1 简介	1
1.2 开放源码软件的革命	1
1.3 Python 的历史	2
1.4 Python 模块	3
1.5 Python 和本书的一般注意事项	3
1.6 本书导读	3
1.7 因特网和万维网资源	8
第 2 章 Python 编程概述	9
2.1 简介	9
2.2 第一个 Python 程序：打印一行文本	9
2.3 修改第一个 Python 程序	11
2.4 另一个 Python 程序：整数求和	12
2.5 内存概念	14
2.6 算术运算	15
2.7 字符串格式化	19
2.8 做出决策：相等运算符和关系运算符	21
2.9 缩进	24
2.10 对象思想：对象技术简介	25
第 3 章 控制结构	27
3.1 概述	27
3.2 算法	27
3.3 伪代码	27
3.4 控制结构	28
3.5 if 选择结构	29
3.6 if/else 和 if/elif/else 选择结构	30
3.7 while 重复结构	34
3.8 算法陈述：案例分析 1（由计数器控制的重复）	35
3.9 算法陈述，自上而下求精法：案例分析 2（由哨兵值控制的重复）	37
3.10 算法陈述，自上而下求精法：案例分析 3（嵌套控制结构）	40
3.11 增量赋值符号	43
3.12 由计数器控制的重复的本质	44
3.13 for 重复结构	45
3.14 使用 for 重复结构	47
3.15 break 和 continue 语句	49
3.16 逻辑运算符	50
3.17 结构化编程总结	53
第 4 章 函数	57
4.1 概述	57
4.2 Python 中的程序组件	57

4.3	函数	58
4.4	math 模块的函数	58
4.5	函数定义	60
4.6	随机数生成	62
4.7	示例：博彩游戏	63
4.8	作用域规则	65
4.9	关键字 import 和命名空间	68
4.10	递归	70
4.11	递归示例：斐波拉契序列	72
4.12	递归与重复	74
4.13	默认参数	74
4.14	关键字参数	75
第 5 章	列表、元组和字典	77
5.1	概述	77
5.2	序列	77
5.3	创建序列	79
5.4	使用列表和元组	80
5.5	字典	86
5.6	列表和字典方法	88
5.7	引用和引用参数	92
5.8	将列表传给函数	92
5.9	列表排序和搜索	93
5.10	多下标序列	95
第 6 章	公共网关接口 (CGI) 入门	99
6.1	概述	99
6.2	客户和 Web 服务器交互	99
6.3	简单的 CGI 脚本	103
6.4	向 CGI 脚本发送输入	108
6.5	用 XHTML 表单发送输入并用 cgi 模块获取表单数据	110
6.6	用 cgi.FieldStorage 读取输入	113
6.7	其他 HTTP 标头	114
6.8	示例：交互式门户网站	114
6.9	因特网和万维网资源	117
第 7 章	基于对象的编程	118
7.1	概述	118
7.2	用类实现一个 Time 抽象数据类型	118
7.3	特殊属性	121
7.4	控制属性访问	122
7.5	为构造函数使用默认参数	128
7.6	析构函数	131
7.7	类属性	131
7.8	合成：对象引用作为类成员使用	133
7.9	数据抽象和信息隐藏	135
7.10	软件重用性	136

第 8 章 自定义类	138
8.1 概述	138
8.2 自定义字符串表示: <code>__str__</code> 方法	138
8.3 自定义属性访问	140
8.4 运算符重载	142
8.5 运算符重载的限制	143
8.6 重载一元运算符	144
8.7 重载二元运算符	144
8.8 重载内建函数	145
8.9 类型转换	146
8.10 案例分析: Rational 类	146
8.11 重载序列运算	152
8.12 案例分析: SingleList 类	152
8.13 重载映射运算	156
8.14 案例分析: SimpleDictionary 类	156
第 9 章 面向对象编程: 继承	159
9.1 概述	159
9.2 继承: 基类和派生类	160
9.3 创建基类和派生类	161
9.4 在派生类中覆盖基类方法	164
9.5 继承的软件工程学	165
9.6 合成与继承	166
9.7 “使用”和“知道”关系	166
9.8 案例分析: Point, Circle 和 Cylinder	167
9.9 抽象基类和具体类	170
9.10 案例分析: 继承接口和实现	170
9.11 多态性	173
9.12 类和 Python 2.2	174
第 10 章 图形用户界面组件 (一)	188
10.1 概述	188
10.2 Tkinter 简介	189
10.3 简单的 Tkinter 例子: Label 组件	190
10.4 事件处理模型	192
10.5 Entry 组件	192
10.6 Button 组件	195
10.7 Checkbutton 和 Radiobutton 组件	197
10.8 鼠标事件处理	201
10.9 键盘事件处理	205
10.10 布局管理器	207
10.11 洗牌和发牌模拟	213
10.12 因特网和万维网资源	215
第 11 章 图形用户界面组件 (二)	216
11.1 概述	216
11.2 Pmw 简介	216
11.3 ScrolledListBox 组件	216

11.4	ScrolledText 组件.....	218
11.5	MenuBar 组件.....	220
11.6	弹出菜单.....	223
11.7	Canvas 组件.....	225
11.8	Scale 组件.....	226
11.9	其他 GUI 工具包.....	227
第 12 章	异常处理.....	229
12.1	概述.....	229
12.2	引发异常.....	229
12.3	异常处理.....	230
12.4	示例: DivideByZeroError.....	232
12.5	Python 的 Exception 层次结构.....	234
12.6	finally 子句.....	235
12.7	Exception 对象和跟踪.....	238
12.8	程序自定义异常类.....	240
第 13 章	字符串处理和正则表达式.....	243
13.1	概述.....	243
13.2	字符和字符串基础.....	243
13.3	字符串表示.....	245
13.4	搜索字符串.....	246
13.5	连接和分解字符串.....	247
13.6	正则表达式.....	248
13.7	编译正则表达式和处理正则表达式对象.....	249
13.8	正则表达式的重复和置位字符.....	250
13.9	字符类和特殊序列.....	252
13.10	正则表达式的字符串处理函数.....	254
13.11	分组.....	255
13.12	因特网和万维网资源.....	256
第 14 章	文件处理和序列化.....	257
14.1	概述.....	257
14.2	数据层次结构.....	257
14.3	文件和流.....	258
14.4	创建顺序访问文件.....	259
14.5	从顺序访问文件读取数据.....	261
14.6	更新顺序访问文件.....	265
14.7	随机访问文件.....	265
14.8	模拟随机访问文件: shelve 模块.....	266
14.9	将数据写入 shelve 文件.....	266
14.10	从 shelve 文件获取数据.....	267
14.11	示例: 一个事务处理程序.....	268
14.12	对象序列化.....	271
第 15 章	可扩展标记语言 (XML).....	274
15.1	概述.....	274
15.2	XML 文档.....	274
15.3	XML 命名空间.....	277

15.4	文档对象模型 (DOM)	280
15.5	Simple API for XML (SAX)	280
15.6	文档类型定义 (DTD)、架构和验证	281
15.7	XML 词汇表	287
15.8	可扩展样式表语言 (XSL)	292
15.9	因特网和万维网资源	296
第 16 章	Python 的 XML 处理	298
16.1	概述	298
16.2	动态生成 XML 内容	298
16.3	XML 处理包	300
16.4	文档对象模型 (DOM)	301
16.5	用 xml.sax 解析 XML	307
16.6	案例分析: 用 Python 和 XML 实现论坛	309
16.7	因特网和万维网资源	321
第 17 章	数据库应用程序编程接口 (DB-API)	322
17.1	概述	322
17.2	关系数据库模型	322
17.3	关系数据库简介: Books 数据库	323
17.4	结构化查询语言 (SQL)	327
17.5	Python DB-API 规范	338
17.6	数据库查询示例	338
17.7	查询 Books 数据库	341
17.8	读取、插入和更新数据库	344
17.9	因特网和万维网资源	348
第 18 章	进程管理	349
18.1	概述	349
18.2	os.fork 函数	349
18.3	os.system 函数和 os.exec 函数家族	355
18.4	控制进程的输入和输出	358
18.5	进程间通信	361
18.6	信号处理	363
18.7	发送信号	364
第 19 章	多线程处理	367
19.1	概述	367
19.2	线程状态: 生命期	367
19.3	threading.Thread 示例	369
19.4	线程同步	371
19.5	生产者/消费者关系: 无线程同步	372
19.6	生产者/消费者关系: 有线程同步	376
19.7	生产者/消费者关系: Queue 模块	380
19.8	生产者/消费者关系: 循环缓冲区	383
19.9	信号机	388
19.10	事件	390
第 20 章	联网	392
20.1	概述	392

20.2	通过 HTTP 定址 URL.....	392
20.3	建立简单服务器（使用流套接字）.....	394
20.4	建立简单客户（使用流套接字）.....	395
20.5	通过流套接字连接进行客户/服务器交互.....	396
20.6	通过数据文报进行无连接的客户/服务器交互.....	399
20.7	使用多线程服务器的客户/服务器 Tic-Tac-Toe 游戏.....	401
第 21 章	安全性.....	409
21.1	概述.....	409
21.2	密码系统古今谈.....	409
21.3	加密密钥.....	412
21.4	公钥加密.....	414
21.5	密码破解.....	415
21.6	密钥协商协议.....	416
21.7	密钥管理.....	416
21.8	数字签名.....	417
21.9	公钥基础结构.....	418
21.10	安全协议.....	420
21.11	身份验证.....	422
21.12	安全攻击.....	424
21.13	运行受限 Python 代码.....	427
21.14	网络安全.....	430
21.15	隐写术.....	432
第 22 章	数据结构.....	434
22.1	概述.....	434
22.2	自引用类.....	434
22.3	链表.....	434
22.4	堆栈.....	441
22.5	队列.....	443
22.6	树.....	444
第 23 章	案例分析：网上书店.....	449
23.1	概述.....	449
23.2	HTTP 会话和会话跟踪技术.....	449
23.3	在网上书店中跟踪会话.....	450
23.4	网上书店体系结构.....	453
23.5	配置网上书店.....	455
23.6	进入网上书店.....	456
23.7	从数据库获得书籍列表.....	457
23.8	查看一本书的详细资料.....	462
23.9	在购物车中添加商品.....	465
23.10	查看购物车.....	466
23.11	结账.....	470
23.12	处理订单.....	472
23.13	错误处理.....	473
23.14	处理无线客户端（XHTML Basic 和 WML）.....	475
23.15	因特网和万维网资源.....	494

第 24 章 多媒体.....	495
24.1 概述.....	495
24.2 PyOpenGL 简介.....	495
24.3 PyOpenGL 示例.....	495
24.4 Alice 简介.....	501
24.5 狐狸、鸡和种子问题.....	501
24.6 pygame 简介.....	505
24.7 Python CD Player.....	506
24.8 Python Movie Player.....	510
24.9 用 pygame 开发太空船游戏.....	513
24.10 因特网和万维网资源.....	524
第 25 章 Python 服务器页 (PSP)	525
25.1 概述.....	525
25.2 Python Servlet.....	525
25.3 PSP 简介.....	526
25.4 第一个 PSP 示例.....	527
25.5 隐式对象.....	529
25.6 脚本编程.....	529
25.7 标准动作.....	532
25.8 预编译指令.....	540
25.9 案例分析: 用 Python 和 XML 实现论坛.....	545
25.10 因特网和万维网资源.....	559
附录 A Python 开发环境.....	560
A.1 概述.....	560
A.2 集成开发环境: IDLE.....	560
A.3 其他集成开发环境.....	564
A.4 因特网和万维网资源.....	566
附录 B Python 2.2 的其他特点.....	567
B.1 概述.....	567
B.2 迭代器.....	567
B.3 生成器.....	574
B.4 嵌套作用域.....	577
B.5 因特网和万维网资源.....	579

第1章 绪 论

学习目标

- 了解开放源码软件
- 熟悉 Python 程序语言的历史
- 本书导读

1.1 简介

欢迎进入 Python 的世界！希望通过我们艰苦的努力，带给读者一本内涵丰富、寓教于乐的计算机参考书。为此，我们采用了多种方法，最终写成了一本与众不同的 Python 参考书。例如，我们很早便讲解了 Python 如何与公共网关接口（CGI）配合，以便编写基于 Web 的应用程序。这样一来，便可在本书剩余部分，更好地演示大量动态的、基于 Web 的应用程序。本书介绍了大量重要主题，包括面向对象编程（OOP）、Python 数据库应用程序编程接口（DB-API）、图形、可扩展标记语言（XML）以及安全性。不管您是一名新手还是有经验的程序员，本书提供的信息量、趣味性和挑战性，都会让您称心如意。

本书面向所有层次的读者，从正式的程序员，一直到很少或根本没有编程经验的自学者。同一本书，怎么可能同时适用于高低两个层次的读者呢？其中的关键在于，我们以成熟的“结构化编程”以及“基于对象的编程”技术为准，始终都在强调如何编写“思路清晰”的程序。非程序员出身的人可以学会基本的技能，为将来进行良好编程奠定基础。有经验的程序员将获得对语言的严谨解释，而且书中的内容有助于改进他们的编程风格。为帮助刚入门的程序员，我们采用一种清晰的、平铺直叙的方式，其间穿插大量插图。另外，更重要的是，本书提供了数百个功能完整的 Python 程序。本书所有示例程序都可从我们的网站（www.deitel.com）下载。

大多数人或多或少都熟悉计算机令人激动的功能。使用本书，可学会如何指挥计算机，亲自实现那些功能。毕竟，是“软件”（要求计算机采取行动和做出决策的指令）在控制着计算机（通常称为“硬件”）。

今天，几乎每个领域都越来越依赖计算机。在其他成本都在稳定、缓慢提升的同时，计算成本却呈显著下降态势——这完全归功于硬件和软件技术的快速发展。25~30 年前，需要几个大房间才能摆下的大型计算机，以及那些动辄数百万美元的“巨无霸”，现在只需指甲大小的硅芯片即可搞定。成本也降至每片几美元左右。不过，具有讽刺意味的是，“硅”是我们这个地球上不值钱的东西之一。在海边随手抓一把沙子，里面含的绝大多数元素便是“硅”。硅芯片技术的问世，使得计算成本变得异常低廉，也直接促成了如今数亿台计算机广泛应用于各行各业。在商业、工业、政府以及我们的个人生活方面，计算机都能提供强有力的帮助。而且再过几年，这个数字还可以轻松翻一倍。

从现在起，您将开始一段美妙的、令人激动的、充满挑战的以及令人回味无穷的学习之旅。在这个旅程中，如果您碰到什么问题，不妨发信给 deitel@deitel.com，或浏览我们的网站（www.deitel.com、www.prenhall.com/deitel 和 www.informIT.com/deitel）。祝您学习顺利。

1.2 开放源码软件的革命

如果程序的源代码免费提供给任何开发人员进行修改，传播，以及用作其他软件的基础，就称为“开放源码软件”或“开源软件”。¹相反，“封闭源码软件”则禁止其他开发者以之为基础开发其他软件。

开放源码并不是一个新概念。早在 20 世纪 60 年代，开源技术就是现代计算工业得以迅速发展的一项主要推动力。特别是美国政府出资兴建了今天 Internet 的前身，并鼓励计算机科学家开发各种各样的

¹ 真正的开源（Open-Source）软件要符合 9 条要求。详情参见 www.opensource.org/docs/definition.html。

技术，以实现在各种计算机平台上的分布式计算。这演变成了今天的一系列重要技术，比如用于实现 Internet 通信的协议。Internet 建好之后，封闭源码技术及软件才逐渐在软件工业中流行起来，开放源码的思想在 20 世纪 80 年代和 90 年代初一度受到抑制。但在这之后，为了反击大多数商业软件的“封闭”本质，并因为封闭源码厂商冷淡的态度，开放源码软件再度流行。如今，Python 已成为快速成长的开源软件社区的一部分，其他还有 Linux 操作系统、Perl 脚本编程语言、Apache Web 服务器以及成百上千的其他软件项目。

计算机行业的一些人将开放源码视为 Free（免费）软件。大多数情况下，这种说法是成立的。不过，在谈到开放源码软件“Free”时，一种更合适的说法是“Freedom”（自由）——任何开发者都可自由修改源码，交流编程思想，参与软件开发过程，以及基于现有的开源软件开发出新的软件程序。大多数开源软件实际都是有版权的，要使用它们，需要获得相应的许可证。开放源码许可协议所规定的条款是各不相同的；有的只有极少限制（比如 Artistic License），另一些可能有许多限制，详细规定软件的修改和使用方式。通常，软件的版权由个人开发者或者一个组织所持有。若要查看许可证/许可协议的一个例子，可访问 www.python.org/2.2/license.html，仔细阅读 Python 的使用许可协议。

通常，可通过 Internet 下载开源产品的源代码。这使开发者能够学习、检查和修改源码，以满足他们自己的需求。由于有整个开发者社区作为后盾，有许多人审查代码，所以相较于封闭源码软件开发，性能和安全问题能更快得到检测 and 解决。另外，更大的开发者社区可为一个软件提供更多的特性。通常，代码修正数小时内便可推出，开源软件新版本的推出频率也要比封闭源软件高出许多。在开源软件的使用许可协议中，通常要求开发者发表他们所做的任何改进，这使得开源社区不断地发展壮大，不断地改进现有的产品。例如，Python 开发者喜欢加入 comp.lang.python 新闻组，交流有关 Python 开发的心得体会。Python 开发者还可为自己的修改变写文档，并通过 Python Enhancement Proposals (PEPS) 提交给 Python Software Foundation (Python 软件基金会)。这样一来，Python 开发团体就能对提议的修改进行评估，并在未来的版本中集成好的改进。

许多公司（比如 IBM、Red Hat 和 Sun）都支持开源开发者及项目。有时，这些公司还会取得开源应用程序，并通过商业途径销售它们（这取决于软件的使用许可协议）。为获得赢利，他们还还为开源软件提供一系列服务，比如技术支持、为客户定做软件 and 培训等等。开发者可作为顾问或培训者提供服务，帮助用户实现软件。欲知开源软件的详情，请访问 Open Source Initiative 网站 (www.opensource.org)。

1.3 Python 的历史

Python 起源于 1989 年末。当时，CWI（阿姆斯特丹国家数学和计算机科学研究所）的研究员 Guido van Rossum 需要一种高级脚本编程语言，为其研究小组的 Amoeba 分布式操作系统执行管理任务。为创建新语言，他从高级教学语言 ABC（All Basic Code）汲取了大量语法，并从系统编程语言 Modula-3 借鉴了错误处理机制。然而，ABC 的一个重大缺点是扩展性不足；语言不是开放式的，不利于改进或扩展。因此，Van Rossum 决定在新语言中合成来自现有语言的许多元素，但要求必须能通过类和编程接口进行扩展。他将这种新语言命名为 Python（原意为“大蟒蛇”）——来源于 BBC 当时正在热播的喜剧片连续剧“Monty Python”。

自 1991 年初公开发售后，Python 开发者和用户社区逐渐壮大，Python 语言逐渐演变成一种成熟的、并获得良好支持的程序语言。Python 被用来开发大量应用程序，从创建网上电子邮件程序到控制水下交通工具，以及配置操作系统和创建动画片等等。2001 年，核心 Python 开发团队移师 Digital Creations 公司，后者是 Zope（用 Python 编写的一个 Web 应用程序服务器）的创始人。预计 Python 会继续成长与发展，进入一个全新的领域。

1.4 Python 模块

Python 是一种模块化的可扩展语言，它可随时集成新“模块”(Modules)——一种可重用的软件组件。任何 Python 开发人员都能编写新模块来扩充 Python 的功能。Python 源代码、模块和文档资料的主要“集散地”是 Python 网站 (www.python.org)，该网站还计划建立一个专门维护 Python 模块的网站。

1.5 Python 和本书的一般注意事项

Python 经过了良好的设计，无论新手还是有经验的程序员都能快速学习和理解这种语言，并轻松上手。和其前身不同，Python 具有良好的移植和扩展能力。Python 的语法和设计有利于养成良好的编程习惯，并可在不牺牲程序扩展性与维护性的同时，显著缩短开发时间。

Python 相当简单，新手程序员可轻松上手；但它同时具有强大的功能，对专家也有足够的吸引力。本书通过丰富、完整且实际有效的例子和讨论，介绍了大量编程概念。随着学习的深入，读者可通过我们创建的实际应用程序，探索更加复杂的主题。贯穿全书，我们始终在强调良好的编程习惯，并给出大量移植性提示以及解释如何防范常见的编程错误。

Python 是当前移植能力最强的程序语言之一。最初，它是在 UNIX 上实现的。但之后扩展到了其他许多平台，其中包括 Microsoft Windows 和 Apple Mac OS X。Python 程序通常可直接从一种操作系统移植到另一种操作系统，无需任何更改，而且能确保正确执行。

1.6 本书导读

本节简要介绍全书讲解的各个主题。有的章末附有“因特网和万维网资源”小节，提供有关 Python 编程的更多信息。

第 1 章——绪论：介绍开放源码革命，讲述 Python 程序语言的起源，并概括本书其余各章主要内容。

第 2 章——Python 编程概述：介绍一个典型的 Python 编程环境，并解释了 Python 程序的基本语法。我们讨论了如何从命令行运行 Python。除解释器之外，Python 还可在一个交互模式中执行语句，即输入一个语句，立即执行一个。在本章及全书，我们会展示许多交互会话，强调各种平常容易忽视的编程要点。本章讨论了变量，介绍了算术运算、赋值、相等、关系和字符串运算符。我们介绍了决策和算术运算。字符串是一种基本的、功能强大的内建数据类型。我们介绍了一些标准的输出格式化技术，并讨论了“对象”和“变量”的概念。对象是值的容器，而变量是用于引用对象的名称。结束本章的学习后，读者将理解如何编写简单而又完整的 Python 程序。

第 3 章——控制结构：介绍用于解决问题的“算法”(过程)。解释了高效率使用控制结构的重要性：控制结构可使程序易于理解、调试和维护，而且有助于首次试运行即告成功。本章介绍了选择结构 (if, if/else 和 if/elif/else) 和重复结构 (while 和 for)。我们详尽解释了如何进行重复，并对比了计数器控制的循环和哨兵值控制的循环。同时还解释了“自上而下求精法”为什么有助于生成结构正确的程序，有助于建立一种流行的程序设计辅助手段——伪代码。本章提供的案例分析演示了如何快速方便地将伪代码算法转换成能实际工作的 Python 代码。本章还解释了用于改变控制流程的 break 和 continue 语句。另外，我们展示了怎样用逻辑运算符 and, or 和 not 在程序中做出复杂的决策。本章的几个交互式会话演示了如何创建一个 for 结构，以及如何避免结构化编程中的一些常见编程错误。本章最后对结构化编程进行了总结。利用本章介绍的技术，能在任何程序语言中有效使用控制结构，而非局限于 Python。本章有助于读者培养良好的编程习惯，为进行本书后面更高级的编程任务打好基础。

第 4 章——函数：讨论了“函数”的设计与构建。在 Python 中，和函数相关的功能包括内建函数、程序员定义的函数和递归等。本章介绍的技术是创建具有正确结构的程序的基础（尤其是系统程序员和

应用程序开发者针对实际环境而开发的较大的程序和软件时)。本章展示了“分而治之”策略,它是解决复杂问题的有效手段,具体做法是将这些问题分解成较简单的交互式组件。我们首先将模块描述成一种容器,它用于容纳一组有用的函数。我们介绍了 `math` 模块,并讨论了其中许多同数学有关的函数。许多人喜欢学习随机数和模拟,在掷骰子游戏案例分析(其中充分运用了控制结构)过程中,他们会觉得非常有趣。本章解释了如何解决一个斐波拉契和阶乘问题,具体利用的是一种名为“递归”的技术(即函数调用自身)。本章讨论了作用域规则,并用一个例子来检查局部变量和全局变量。本章还讨论了程序如何采取各种不同的方式导入模块及其元素, `import` 语句会对程序的命名空间造成什么影响。Python 的函数可指定默认参数和关键字参数。我们讨论了向函数传递信息的两种方式,并在一个交互式会话中讲解了某些常见的编程错误。

第 5 章——列表、元组和字典:本章详细介绍了 3 种高级的 Python 数据类型,即列表、元组和字典。利用这些数据类型,Python 程序员可通过最少的代码行完成非常复杂的任务。字符串、列表和元组全都属于“序列”的范畴,后者是一种数据类型,可通过索引和“切片”进行处理。我们讨论了怎样创建、访问和操纵序列,并提供了一个例子(它根据由值构成的一个序列来创建柱形图)。我们介绍了列表和元组用于 Python 程序的各种方式。字典是“可映射”类型——键和值成对保存,并相互映射(对应)。我们通过一个用于存储学生成绩的例子解释了怎样创建、初始化和操纵字典。另外还介绍了“方法”,它们是用于执行对象(比如列表和字典)的各种操作的函数,并解释了如何利用方法来访问、排序和搜索数据。这些方法可轻松地完成各项计算任务;换用其他语言,则可能要用大量代码行才可获得同样的效果。我们还介绍了“不可变序列”和“可变序列”。如果将多个可变序列传递给函数,会产生一个重要的、可能出乎预料的“副作用”。我们用一个例子来展示这种副作用的效果。

第 6 章——公共网关接口 (CGI) 入门:本章介绍了可使应用程序 (CGI 程序或脚本) 与 Web 服务器进行交互的一种协议,即“超文本传输协议”(HTTP),它是 Web 服务器和 Web 浏览器之间进行数据通信的一种基本组件。我们解释了客户如何通过 Internet 连接到服务器,以及运行 CGI 程序的 Web 服务器如何将响应发送给客户。从 Web 服务器发送给 Web 浏览器的最常见的数据就是网页,它是用“可扩展超文本标记语言”(XHTML) 格式化的一种文档。在本章,将介绍如何创建简单 CGI 脚本。另外,还介绍了如何将用户输入从浏览器发送给一个 CGI 脚本,并用实例展示了如何在 Web 浏览器中显示一个人的姓名。然后重点讲解怎样利用 XHTML 表单,将用户输入发送给一个 CGI 脚本,以及在客户端和服务端上的 CGI 程序之间传递数据。随后演示如何用 `cgi` 模块处理表单数据。本章描述了随 CGI 使用的各种 HTTP 标头。最后,我们在一个 Web 门户案例分析中运用了 CGI 知识,它允许用户登录到一个虚拟旅游网站,查看特价信息。

第 7 章——基于对象的编程:本章开始讨论基于对象的编程。这是讲解“数据抽象”的良机——利用的是一开始就设计为面向对象的 Python 语言。近几年,数据抽象已成为计算机初级课程的重要主题之一。我们讨论了如何通过一个类实现抽象数据类型,如何初始化和访问类的成员。和其他语言不同,Python 不允许程序员禁止属性访问。本章和第 8 章及第 9 章讨论了几种访问控制技术。我们介绍了“私有”属性以及 `get` 和 `set` 方法,它们控制着对数据的访问。所有对象和类都具有公共属性,本章讨论了它们的名称和值。讨论了默认构造函数后,我们进一步扩展了以前的例子。另外还介绍了用于指出错误的 `raise` 语句。类可包含类属性——即只需创建一次,然后可供类的所有实例使用的数据。还讨论了一个有关合成的例子,其中的实例可将其他实例作为数据成员引用。本章最后讨论了软件重用问题。

第 8 章——自定义类:本章讨论 Python 用于自定义类行为的几个方法。这些方法扩展了以前各章介绍的访问控制机制。自定义技术最强大的功能或许就是“运算符重载”,它允许程序员告诉 Python 解释器如何将现有的运算符用于新类型的对象。Python 已知如何将运算符用于内建类型的对象,比如整数、列表和字符串等。但是,假如创建了一个新的 `Rational` 类,那么在 `Rational` 对象之间使用的一个加号 (+) 到底表示什么呢?在本章,程序员将学习如何“重载”加号。这样一来,如果在表达式的两个 `Rational` 对象之间使用加号,解释器就会生成对一个“运算符方法”的方法调用,它负责将两个 `Rational` 对象“加”到一起。本章讨论了运算符重载的基础知识及其存在的限制,如何重载一元运算符和二元运算符,以及如何在不同类型之间转换。本章还讨论了如何自定义类,使其具有列表或字典行为。

第9章——面向对象编程：继承：本章介绍了面向对象程序语言最基本的能力之一，即“继承”。继承是软件重用的一种形式，通过吸收现有类的功能，再添加合适的新功能，可以快速和简单地开发出新类。本章讨论了基类和派生类的记号法、直接基类、间接基类、基类和派生类中的构造函数和析构函数以及继承的软件工程学问题。本章还比较了各种面向对象关系，比如继承与合成。通过继承，人们发展出了 Python 最值得骄傲的一种编程技术，即“多态性”。如果许多类都通过继承和同一个公共基类联系，那么每个派生类对象都可视为一个基类实例。这样一来，程序可采用一种泛型方式编写，独立于派生类对象的具体类型。新的对象可由同一个程序处理，使系统更容易扩展。人们经常采取这种形式的编程来实现当今流行的“图形用户界面”（GUI）。本章最后讨论了 Python 2.2 支持的新的面向对象编程技术。

第10章——图形用户界面组件（一）：本章介绍了 Tkinter，该模块为流行的 Tcl/Tk 图形用户界面工具包提供了一个 Python 接口。本章在简要概述 Tkinter 模块之后，做了一番概述。使用 Tkinter，可快速方便地创建图形化程序。本章介绍了几个基本 Tkinter 组件：Label, Button, Entry, Checkbutton 和 Radiobutton。我们讨论了事件处理的概念，它是 GUI 编程的核心；本章还通过几个例子揭示如何在 GUI 应用程序中处理鼠标及键盘事件。本章最后深入探讨了 Pack, Grid 和 Place 这几种 Tk 布局管理器。结束本章的学习后，读者应能理解大多数 Tkinter 应用程序。

第11章——图形用户界面组件（二）：本章讨论了附加的 GUI 编程主题。我们介绍了 Pmw 模块，它对基本的 Tk GUI 元件集进行了扩展。本章展示了如何创建菜单、弹出菜单、滚动文本框和窗口。本章的例子演示了如何将文本从一个窗口拷贝到另一个，允许用户选择和显示图片，更改文本字体以及更改窗口背景颜色。最有趣的是，一个长度仅 39 行的程序，它允许用户用鼠标在 Canvas 组件上绘图。本章最后讨论了适用于 Python 程序员的其他 GUI 工具包，其中包括 pyGTK, PyOpenGL 以及 wxWindows。在本书剩余部分，大量例子都要用到第 10 章和第 11 章讲解的 GUI 技术。结束这两章的学习后，读者就可着手为数据库、联网和简单游戏程序编写其中的 GUI 部分。

第12章——异常处理：利用本章提供的知识，程序员可使自己编写的程序可靠性和容错性更佳、更适合用于面向关键任务和事务的环境。本章首先解释了异常处理技术，然后讨论了应在什么时候进行异常处理，还通过在一个例子中使用 try/except/else 语句（该例子能非常好地处理“除以零”这一严重逻辑错误），从而介绍了异常处理的基础知识。程序员可用 raise 语句明确引发异常；我们讨论了该语句的语法，并演示了它的用法。本章解释了如何从异常提取信息，以及以何种方式在何时引发异常。我们解释了 finally 语句，详细解释了应在什么时候、在什么地方捕捉异常。在 Python 中，异常的本质是“类”。我们通过探讨异常层次结构，解释如何创建自定义异常，阐述了异常和类的关系。本章最后提供了一个例子，它利用 traceback 模块来探讨 Python 异常的本质及内容。

第13章——字符串处理和正则表达式：本章解释了如何处理字符串外观、顺序和内容。字符串是大多数 Python 输出的基础。本章讨论了 count, find 和 index 方法，它们用于在字符串中搜索子字符串。split 方法可将字符串分解成一个字符串列表。replace 方法可将字符串中的一个子字符串替换成另一个。这些方法提供了基本的文字处理能力，但程序员经常需要进行更强大的、基于模式的文字处理。re 正则表达式模块在 Python 中实现了基于模式的文字处理。正则表达式处理是一个非常复杂的主题，而且存在许多缺陷。我们从基本正则表达式开始，一直讲到较高级的一些主题。我们指出了最常见的编程错误，并用例子解释这些错误是如何发生的，以及如何避免。我们讨论了 re 模块的常用函数和类，并介绍了常用的正则表达式元字符和序列。我们演示了分组的概念，它允许程序员从正则表达式处理结果中提取信息。Python 正则表达式可进行编译以增强正则表达式的处理性能，所以我们讨论了何时适合这样做。

第14章——文件处理和序列化：本章讨论了用于处理顺序访问和随机访问文本文件的技术，并根据位、字节、字段、记录和文件，对数据层次结构进行了概述。然后，我们介绍了 Python 的简单文件及文件句柄视图。我们用示例程序展示了如何打开和关闭文件，如何将数据顺序存储到一个文件中，以及如何从一个文件中顺序读取文件，由此讨论了顺序访问文件的详情。这些例子使用第 13 章介绍的字符串格式化技术输出从一个文件中读取的数据。本章还提供了一个比较实用的信用查询程序，可从一个顺序访问文件中获取数据，然后根据获取的数据，对输出进行格式化。本章讨论了如何使用 print 语句将文本重定向至任何一个文件，其中包括程序用于显示错误消息的“标准错误文件”。讨论随机访问文件时，我们

使用 `shelve` 创建一个随机访问文件，并在该文件中读写数据。最后展示了一个较大的事务处理程序，它综合运用了本章讨论的所有技术。Python 高级数据类型的一个优点是程序可对任意 Python 对象进行序列化（存储到磁盘）。我们在一个例子中使用 `cPickle` 模块将 Python 字典保存到磁盘，以便将来使用。

第 15 章——可扩展标记语言 (XML)：XML 是用于创建标记语言的一种语言。HTML 用于格式化要显示的信息；相反，XML 用于对信息进行结构化。它不像 HTML 那样有一套固定标记，它允许文档作者自行创建新标记。本章概要介绍了“解析器”（对 XML 文档及其数据进行处理的程序）。另外还解释了“良构文档”（即符合语法规则的文档）的要求。随后介绍了命名空间（用于区分同名元素）。还介绍了文档类型定义 (DTD) 文件和 Schema（架构）文件，它们通过指定元素在 XML 文档中的类型、顺序、数量和属性，为 XML 文档提供了一个结构化定义。通过定义一个 XML 文档的结构，DTD 或 Schema 可减少使用该文档的那个应用程序的验证和错误检查工作。本章介绍了一种非常流行的与 XML 相关的技术，名为“可扩展样式表语言”(XSL)，它的作用是将 XML 文档转换成另一种文档格式，比如 XHTML。本章只是对 XML 的概述。第 16 章要进一步介绍如何在 Python 中处理 XML。

第 16 章——Python 的 XML 处理：本章要讨论如何使用标准和第三方模块，简单但高效地进行 Python XML 处理和操纵。本章概述了处理 XML 文档的几种方式，并讨论了 W3C 的文档对象模型 (DOM)，它是用于 XML 的一种应用程序编程接口 (API)，与平台和语言无关。DOM API 提供了一系列标准接口（即方法、对象等等），可用它们处理一个 XML 文档的内容。XML 文档具有层次结构，所以 DOM 将 XML 文档表示成树结构。使用 DOM，程序可动态修改文档的内容、结构和格式。本章还介绍了除 DOM 之外的另一种选择，名为 Simple API for XML (SAX)。DOM 是在内存中构建一个树结构；SAX 则不同，它在文档中遇到起始标记、结束标记、属性等等时，会调用特定的方法。因此，通常将 SAX 称为“基于事件的 API”。Python 的 XML 支持是借助 `xml.dom.ext` (DOM) 以及 `xml.sax` (SAX) 模块来实现的。本章使用了 4Suite（由 FourThought 公司开发）和 PyXML，这是 Python XML 模块的两个集合。本章最后提供了一个案例分析，它用 XML 实现了一个基于 Web 的论坛。

第 17 章——数据库应用程序编程接口 (DB-API)：本章介绍如何用程序查询和处理数据库。大多数实用的商业程序和 Web 应用程序都要基于“数据库管理系统”(DBMS)。为支持 DBMS 应用程序，Python 提供了“数据库应用程序编程接口”(DB-API)。本章使用“结构化查询语言”(SQL) 来查询和处理“关系数据库管理系统”(RDBMS)。具体地说来，RDBMS 是一个 MySQL 数据库。为建立与 MySQL 的接口，Python 使用了模块 `MySQLdb`。本章包含 3 个例子：第一个是根据用户指定条件显示作者信息的 CGI 程序；第二个创建了 GUI 程序，允许用户输入 SQL 查询，然后显示查询结果；第 3 个例子是更实用的 CGI 程序，允许用户维护一个联系人列表。可在数据库中添加、删除、更新和查找联系人。

第 18 章——进程管理：本章讨论了“并发性”。大多数程序语言都提供一系列简单控制结构，允许程序员每次执行一个任务，上一个任务结束之后才能执行下一个。这种控制结构不允许采取并发操作。今天，由计算机执行的并发性操作通常以操作系统指令的形式实现，这只适用于有经验的系统程序员。Python 则不然，它使应用程序的开发者也能使用并发性指令。本章介绍了如何使用 `fork` 命令新建进程；如何用 `exec` 和 `system` 命令执行单独的程序；还演示了如何用 `popen` 命令控制输入和输出。注意，有的命令只适合 Unix 平台。本章讨论了 Python 的跨平台能力，并用实例演示如何根据操作系统来执行特定的任务。我们讨论了进程之间的通信方法，其中包括管道和信号。在本章的信号处理例子中，演示了如何判断用户正在试图中断程序；如何指定发生此类事件时要执行的行动。

第 19 章——多线程处理：本章介绍了“线程”，它是“轻量级进程”。相较于第 18 章用 `fork` 等命令创建的全功能进程，线程通常更有效。本章探讨了基本的线程处理概念，其中包括线程生命期中可能出现的各种状态。我们讨论了如何通过子类化 `threading.Thread` 和覆盖 `run` 方法，将线程包括到一个程序中。本章后半部分用一系列例子阐述了经典的生产者/消费者关系。我们为这个问题开发了几个解决方案，并介绍了线程同步和资源分配概念。本章讨论了线程处理控制指令，比如锁、条件变量、信号机和事件等等。最后一个方案用 `Queue` 模块保护对队列所存储共享数据的访问。示例演示了多线程程序存在的危险，以及如何避免这些危险。我们的解决方案还演示了可重用类的巨大价值。我们重用生产者类和消费者类来访问各种同步和非同步的数据类型。结束本章的学习后，读者就可利用许多 Python 工具编写实用的、

可扩展的专业程序。

第 20 章——联网：本章介绍通过计算机网络进行通信的应用程序。Python 等高级语言的一个重要优势在于，可通过较小的、能实际工作的例子来方便地展示及讨论非常复杂的主题。我们讨论了基本联网概念，并提供两个例子：一个 CGI 程序（在浏览器中显示选择的网页）；一个 GUI 程序（在文本区域中显示页面内容）。还讨论了通过套接字进行的客户/服务器通信。示例程序演示了如何利用无连接和基于连接的协议，通过网络来收发消息。本章最具特色的内容就是以“活代码”方式实现一个协作式的客户机/服务器 Tic-Tac-Toe（三连棋）游戏。两个客户与一个多线程的服务器通信，共同玩一个网络版游戏，服务器负责维护游戏的状态。

第 21 章——安全性：本章讨论了 Web 编程的安全问题。Web 编程可快速创建功能强大的应用程序，但这也使计算机易于受到外界攻击。本章讨论了防御性编程技术，它们利用特定技术及工具，帮助程序员防范安全问题。一种工具是加密，我们举例说明了如何用 rotor 模块进行加密和解密，rotor 模块实现了一个置换密码系统。另一个工具是 sha 模块，它用于哈希处理。第 3 个工具是 Python 的限制访问(rexec)模块，它可创建一个受限环境。不受信任的代码将在其中执行，以避免它们损害本地计算机。本章探讨了保证网络安全性的各种技术，比如公钥密码、安全套接字层(SSL)、数字签名、数字证书、数字隐写术和生物测定等等。还讨论了其他类型的网络安全性，比如防火墙和防病毒程序。另外还介绍了一些常见的安全威胁，包括密码破解攻击、病毒、蠕虫和特洛伊木马等。

第 22 章——数据结构：本章讨论 Python 中用于创建和处理标准数据结构的技术。尽管高级数据类型是 Python 内建的，但从概念和编程角度来讨论这些常用数据结构，对读者来说大有益处。本章首先讨论了自引用结构，接着讨论如何创建和维护各种数据结构，包括链表、队列、堆栈和二叉树。我们通过重用链表类来实现队列和堆栈，尽可能减少继承的类的代码，并将讨论的重点集中于代码重用。二叉树类包含了用于进行前序遍历、中序遍历和后序遍历的方法。针对每种类型的数据结构，书中都展示了完整的、可实际工作的程序，并展示了示范输出。

第 23 章——案例分析：网上书店：本章实现了一个网上书店，它综合运用 MySQL，XML 和 XSLT 向不同的客户发送网页。我们首先介绍了 HTTP 会话框架，它可在几个页之间维护客户信息。客户信息在服务器的计算机上进行序列化，供服务器日后使用。然后讨论了 WML，它是无线客户用于在 Web 上传输文档的一种标记语言。尽管我们在演示时用 XHTML，XHTML Basic 和 WML 等客户来访问这个应用程序，但书店系统具有灵活的扩展性，可方便地添加新的客户类型。Python CGI 程序本身不必改变，但程序员可修改网上书店系统，为新客户端创建新的 XML 和 XSLT 文档，从而为新的客户提供服务。网上书店程序确定客户类型，并将恰当的数据发送给客户。本章综合了本书以前许多章的主题，并有力证明了 Python 的一个重要优势——快速而方便地集成多种技术。这些主题包括文件处理、序列化(cPickle 模块)、CGI 表单处理(cgi 模块)、数据库访问(MySQLdb 模块)、XML DOM 和 XSLT 处理(4Suite 模块集)。

第 24 章——多媒体：本章展示了 Python 对多媒体应用程序的支持。对于选修入门级程序课程的学生，本章内容相当重要。一些有趣的多媒体应用程序包括：PyOpenGL（该模块将 Python 绑定到 OpenGL API，创建彩色的、富有吸引力的交互式图形）、Alice（它采取面向对象方式创建和操纵 3D 图形世界）以及 Pygame（它是许多 Python 模块的集合，用于创建跨平台多媒体应用程序，比如交互式游戏等等）。PyOpenGL 示例创建了旋转对象和三维图形。Alice 示例创建了一个图形版本的流行解谜游戏。本例创建的世界中，包括一只狐狸、一只鸡和一粒种子。游戏目标是将它们都移过河，但又不让任何捕食者和被捕食者有机会单独相处。第一个 Pygame 例子合并了 Tkinter 和 Pygame 以创建 GUI 形式的 CD 播放器。第二个例子演示如何播放 MPEG 电影。最后一个 Pygame 例子创建一个电脑游戏，玩家驾驶一艘太空船经过小行星带并收集能源。通过这个例子，我们讨论了图形编程的大量要点和技术。换用其他程序语言，这些项目会变得过于复杂或琐碎，以至于无法放到本书进行讲解。然而，Python 高级的抽象能力、简单的语法以及丰富的模块使不可能的任务变成可能。所有这些令人激动的例子都可放到同一章讲解！

第 25 章——Python 服务器页(PSP)：本章要用熟悉的可扩展超文本标记语言(XHTML)以及 Python 脚本来创建动态 Web 内容。我们分别讨论了客户端和服务端的问题。本章所用的工具包括 Apache 和

Webware for Python, 后者是一套用来创建动态 Web 内容的软件。简要介绍了 Python Servlet 之后, 本章还提供了一些例子, 其中讲解了 PSP 如何处理 Python 独特的缩进样式, 解释了 Scriptlet、动作以及预编译指令。

附录 A——Python 开发环境: 本附录简要介绍了几种 Python 开发环境, 其中包括 IDLE。

附录 B——Python 2.2 的其他特点: 本书出版时正值 Python 2.2 发布。书中反映了大量 Python 2.2 特性, 但仍有一些无法写入正文。因此, 本附录整理了这些附加特性。阅读各章时, 注意参考附录 B, 看看是否有附加的讨论及相应的示例。

网站资源

Deitel 网站 (www.deitel.com) 提供许多 Python 资源, 帮助您在 Windows 或 UNIX/Linux 系统上安装和配置 Python。这些资源包括“Installing Python”, “Installing the Apache Web Server”, “Installing MySQL”, “Installing Database Application Programming Interface (DB-API) modules”, “Installing Webware for Python”以及“Installing Third-Party Modules”。

好了, 就是这么多! 我们经过艰苦工作, 为您创作了这本书。书中含有大量能实际运行的例子、编程提示以及数不清的学习要点, 可帮助您全面地掌握 Python。利用学到的知识, 您可快速和高效地编写基于 Web 的应用程序。阅读本书的过程中, 如果有任何不明白的地方, 或者想报告所发现的错误, 请致函 deitel@deitel.com。我们会尽快答复, 并在 www.deitel.com 公布勘误和其他信息。

Prentice Hall 维护着 www.prenhall.com/deitel, 该网站专门介绍我们为 Prentice Hall 开发的参考书、多媒体软件和基于 Web 的培训产品。可在这里找到针对我们的每本书的“配套网站”, 并提供了包括 FAQ、下载、勘误、更新和白测题在内的大量资源。Deitel & Associates 公司每周为流行的 InformIT 电子刊物撰写一个专栏, 该刊物目前有 80 多万名订阅者。详情请见 www.InformIT.com。

现在, 您将开始一个令人激动的、充满乐趣的 Python 学习之旅, 祝一路顺风!

1.7 因特网和万维网资源

www.python.org

这是寻找 Python 信息的主要地方。Python 主页提供最新新闻、FAQ 以及到 Python 资源的链接, 这些资源包括 Python 软件、教程、用户组和演示等等。

www.zope.com 或 www.zope.org

Zope 是一个可扩展的、开放源码的 Web 应用程序服务器, 它是用 Python 编写的。它由 Digital Creations 公司创建, 整个 Python 开发团队都在这个公司中。

www.activestate.com

ActiveState 为程序员创建开放源码工具。该公司提供的 Python 产品名为 ActivePython 和 Komodo, 这是一个为许多语言 (包括 Python, XML, Tcl 和 PHP 在内) 提供的、开放源码的 IDE (集成开发环境)。ActiveState 为 Windows 平台提供 Python 工具, 并提供了一个名为“Python Cookbook”的 Python 程序集。

homepage.ntlworld.com/tibsnjoan/python.html

提供大量链接, 可通过它们访问正在开发和使用 Python 的许多个人和组织。

www.ddj.com/topics/pythonurl/

“Dr. Dobbs' s Journal”是一份编程出版物, 它提供了一系列有用的 Python 链接。

第 2 章 Python 编程概述

学习目标

- 理解一个典型的 Python 程序开发环境
- 用 Python 编写简单的计算机程序
- 使用简单的输入语句和输出语句
- 熟悉基本数据类型
- 会使用算术运算符
- 理解算术运算的优先顺序
- 会编写简单的、用于做出决策的语句

2.1 简介

Python 为计算机程序设计提供了一种规范方式。本章将介绍 Python 编程，并提供几个例子，演示该语言的重要特性。为了理解每个例子，我们打算每次只分析一个语句。展示了基本概念之后，我们将在第 3~5 章探讨“结构化编程”方式。在探讨 Python 的基本主题的同时，还会提前进行面向对象编程的讨论——这是贯穿全书的一种关键性编程方法。为此，我们专门提供了 2.10 节“对象思想”。

2.2 第一个 Python 程序：打印一行文本

先来看一个能打印一行文本的简单程序。图 2.1 展示了这个程序及其屏幕输出。^①

```
1 # Fig. 2.1: fig02_01.py
2 # Printing a line of text in Python.
3
4 print "Welcome to Python!"
```




图 2.1 文本打印程序

该程序演示了 Python 语言的几项重要特性。让我们分别研究每一行代码。本书展示的每个程序都有行号，以便读者参考。但行号并非实际的 Python 程序的一部分。图 2.1 中，第 4 行是这个程序的重点，它在屏幕上显示短语“Welcome to Python!”。

第 1~2 行都以符号#开头，表明每一行剩余的内容都是“注释”。插入注释的目的是“文档化”程序并改善程序的可读性。注释还可帮助其他程序员阅读和理解您的程序。程序运行时，注释不会导致计算机对其采取任何操作——Python 会忽略注释。我们的每个程序都用一条注释开头，它指出图号以及存储了程序的文件名（第 1 行）。在注释中可使用任何文本。本书所有 Python 程序都可从 www.deitel.com 免费下载。

以符号#开头的一条注释称为“单行注释”，注释会在当前行的末尾结束。以符号#开头的注释也可从一行的中间开始，并一直延续到行末，这样的注释通常用于文档化位于那一行开头的 Python 代码。和其他程序语言不同，Python 没有单独的符号用于多行注释，所以多行注释的每一行都必须以符号#开头。注释文本“Printing a line of text in Python”描述了程序的用途（第 2 行）。

良好编程习惯 2.1 在程序中使用丰富的注释。注释有助于其他程序员理解程序，有助于程序调试（即

^① 本书的资源（包括在 Windows 和 Unix/Linux 平台上安装 Python 的每个步骤的指导）都已发布在 www.deitel.com 网站上。

发现和排除程序中的错误), 并列出有用的信息。以后修改或更新代码时, 注释还有助于您理解自己当初编写的程序。

良好编程习惯 2.2 每个程序都应以一条注释开头, 描述该程序的用途。

第 3 行是空行。程序员使用空行和空格字符使程序易于阅读。空行、空格字符和制表符统称为“空白”或“空距”(White Space), 其中, 空格字符和制表符称为“空白字符”。空行会被 Python 忽略。

良好编程习惯 2.3 加一些空行来增强程序的可读性。

Python 的 print 命令(第 4 行)指示计算机显示包含在引号(")之间的字符串。字符串是一系列字符的组合, 它们包含在一对双引号中。整行称为一个“语句”。在某些程序语言(比如 C++, Java 和 C#)中, 语句必须以分号(;)结尾。在 Python 中, 一旦当前行结束, 大多数语句也就结束了。第 4 行执行时, 会在屏幕上显示“Welcome to Python!”消息。注意, 对字符串进行定界的双引号不会出现在输出中。

Python 中的输出(显示信息)和输入(接收信息)是用字符“流”来完成的。上述语句执行时, 它将字符流“Welcome to Python!”发送给“标准输出流”。标准输出流是应用程序向用户展示信息的一种渠道, 这种信息通常在屏幕上显示, 但也可打印到打印机, 或者写到文件, 甚至可以被朗读出来, 或者输出到盲文设备, 使视力有缺陷的人也能接收输出。

Python 语句可通过两种方式执行。第一种是在编辑器中输入语句以创建一个程序, 再使用.py 扩展名来保存文件(如图 2.1 所示)。Python 文件名通常以.py 结尾, 但也可使用其他扩展名(例如 Windows 平台上的.pyw)。要用 Python 解释器执行文件中的程序, 请在 DOS 或 Unix shell 命令行中输入:

```
python file.py
```

其中的 file.py 就是 Python 文件的名称。shell 命令行是一种文本“终端”, 用户可在其中输入命令, 让计算机系统做出响应。注意, 为了能调用 Python, 系统路径变量必须正确设置, 以包括 python 可执行文件——其中包含了可以运行的 Python 解释器程序。本书的资源(发布在 www.deitel.com 网站上)提供了相应的指导, 帮您设置正确的系统路径变量。

Python 解释器运行存储在文件中的程序时, 会从文件的第一行开始, 并一直执行到文件结束。图 2.1 的输出框显示了 Python 解释器运行 fig02_01.py 的结果。

执行 Python 语句的另一种方式是“交互模式”。在 shell 命令行输入:

```
python
```

即可在“交互模式”中运行 Python 解释器, 它允许程序员直接向解释器输入语句, 每次执行一个语句。

测试和调试提示 2.1 在交互模式输入一个 Python 语句就会执行一个。调试程序时, 这种模式尤其有用。

测试和调试提示 2.2 对一个文件调用 Python 解释器后, 解释器会在文件中的最后一个语句执行之后退出。然而, 如果使用 -i 选项(例如 `python -i file.py`)针对文件调用解释器, 会导致编译器在执行了文件中的语句后进入交互模式。这非常适用于调试程序。

图 2.2 展示了 Python 2.2 在 Windows 上以交互模式运行的样子。前 2 行显示了与所用 Python 的版本信息(2.2b2 表示“版本 2.2, Beta 2”)。第 3 行包含“Python 提示符”(>>>>)。在 Python 提示行输入一个语句, 并按回车键之后, 解释器会执行该语句。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Welcome to Python!"
Welcome to Python!
>>> ^Z
```

图 2.2 交互模式

图 2.2 的第 3 行在屏幕上显示文本“Welcome to Python!”。再次提醒，用于定界的双引号不会打印出来。将文本打印到屏幕后，解释器等待用户输入下一个语句。要退出交互模式，只需输入文件结束符 Ctrl-Z（在 Microsoft Windows 平台上）并按回车键。图 2.3 列出了不同计算机系统上的“文件结束”符。

计算机系统	组合键
UNIX/Linux 系统	Ctrl-D（要在单独一行上）
DOS/Windows	Ctrl-Z（有时要接着按回车键）
Macintosh	Ctrl-D

图 2.3 各种主流计算机系统上的文件结束组合键

2.3 修改第一个 Python 程序

本节通过两个例子继续介绍 Python 编程，它们对图 2.1 中的例子进行了修改，使用多个语句显示单行文本，以及使用单个语句显示多行文本。

2.3.1 用多个语句显示单行文本

“Welcome to Python!”可用几种方式打印。例如，图 2.4 使用两个 print 语句（第 4~5 行）生成和图 2.1 一样的输出。程序的大多数地方都和图 2.1 相同，所以这里只讨论改动的地方。

第 4 行显示字符串“Welcome”。通常，print 语句显示了它的字符串后，Python 会开始一个新行——后续的输出在 print 语句所打印的字符串之后的一行或多行上显示。然而，第 4 行末尾的逗号（,）告诉 Python 不要另起新行，而是在字符串后添加一个空格。因此，程序显示的下一个字符串（第 5 行）会与字符串“Welcome”出现在同一行。

```
1 # Fig. 2.4: fig02_04.py
2 # Printing a line with multiple statements.
3
4 print "Welcome",
5 print "to Python!"
```

```
Welcome to Python!
```

图 2.4 使用几个 print 语句打印单行文本

2.3.2 用单个语句显示多行文本

使用“换行符”，一个语句可显示多行。换行符属于“特殊字符”，用于将屏幕光标定位到下一行的开头。图 2.5 输出 4 行文本，并用换行符决定每个新行何时开始。

```
1 # Fig. 2.5: fig02_05.py
2 # Printing multiple lines with a single statement.
3
4 print "Welcome\nto\n\nPython!"
```

```
Welcome
to

Python!
```

图 2.5 用单个 print 语句打印多行

这里只讨论图 2.5 和图 2.1 及图 2.4 所示程序有区别的地方。第 4 行在屏幕上显示 4 行独立的文本。

通常，字符串中的字符会采取与它们在双引号中一模一样的方式显示。但要注意，有两个字符`\`和`n`没有在输出中出现（它们在第4行出现了3次）。Python 提供了“特殊字符”，可用它们来执行特定的任务，比如退格和回车等等。特殊字符要用反斜杠（`\`）加以区分，反斜杠字符也称为“转义字符”。如果字符串中存在一个反斜杠，它和紧接其后的字符便构成了“转义序列”。`\n` 就是这样的一个转义序列，它代表换行符。`\n` 每出现一次，就会导致屏幕光标（它控制下一个字符的出现位置）移到下一行的开头。要打印一个空行，只需连续使用两个换行符。图 2.6 总结了其他常见的转义序列。

转义序列	说明
<code>\n</code>	换行符。将屏幕光标定位至下一行起始位置
<code>\t</code>	水平制表符。将屏幕光标移至下一个制表位
<code>\r</code>	回车符。将屏幕光标定位至当前行起始位置；不转到下一行
<code>\a</code>	响铃符。发出系统响铃声
<code>\\</code>	反斜杠。用于打印一个反斜杠字符
<code>\"</code>	双引号。用于打印一个双引号字符
<code>'</code>	单引号。用于打印一个单引号字符

图 2.6 转义序列

2.4 另一个 Python 程序：整数求和

下一个程序要求用键盘输入两个整数（如-22，7 和 1024 等），程序计算两者之和，并显示结果。程序调用 Python 函数 `raw_input` 和 `int` 来获得两个整数。同样地，用 `print` 语句显示求和结果。图 2.7 展示了示例程序及其输出。

```

1 # Fig. 2.7: fig02_07.py
2 # Simple addition program.
3
4 # prompt user for input
5 integer1 = raw_input( "Enter first integer:\n" ) # read string
6 integer1 = int( integer1 ) # convert string to integer
7
8 integer2 = raw_input( "Enter second integer:\n" ) # read string
9 integer2 = int( integer2 ) # convert string to integer
10
11 sum = integer1 + integer2 # compute and assign sum
12
13 print "Sum is", sum # print sum

```

```

Enter first integer:
45
Enter second integer:
72
Sum is 117

```

图 2.7 求和程序

第1~2行包含注释，说明程序的编号、文件名和用途。第5行调用 Python 内建函数 `raw_input`，要求用户输入。内建函数是由 Python 提供的一部分代码，用于执行特定任务。任务通过调用函数（函数名，后续一对圆括号`()`）来执行。执行任务后，函数可能返回代表任务结果的值。第4章将深入学习函数，届时会提到其他许多内建函数，并解释如何创建程序员自定义函数。

Python 的 `raw_input` 函数取得一个参数，即`"Enter first integer:\n"`，它请求用户输入。所谓参数（或“实参”）是指函数能接受的一个值，函数要利用它来执行任务。在本例，函数 `raw_input` 接受的是“提示内容”参数（要求用户输入），并将提示内容显示在屏幕上。用户看到提示消息后，需要输入一个数字，并按回车键。这样即可采用字符串形式将数字发送给函数 `raw_input`。

`raw_input` 的结果是一个字符串，其中包含由用户输入的字符，它会被指派给变量 `integer1`，这是用赋值符号=实现的。在 Python 中，可将变量称为“对象”。对象驻留于计算机内存中，并包含程序使用的信息。平时提到“对象”时，通常暗示着有“属性”（数据）和“行为”（方法）同对象联系在一起。对象的方法利用属性来执行任务。变量名（例如 `integer1`）包括字母、数位和下划线（_），但不可用一个数位开头。Python 是一种要区分大小写的语言，例如 `a1` 和 `A1` 属于不同的变量。对象可以有多个名称，即“标识符”。每个标识符（或变量名）都引用（或指向）内存中的一个对象（或变量）。第 5 行的语句通常读作“变量 `integer1` 被指派给 `raw_input("Enter first integer:\n")` 所返回的值”。然而，通过前面的解释，我们知道这行代码实际的含义是“`integer1` 引用了 `raw_input("Enter first integer:\n")` 返回的值”。

良好编程习惯 2.4 有意义的变量名可改善程序的“自编码能力”；也就是说，只需读一读程序，就能轻松理解它，而不必非要阅读手册或使用过多的注释。

良好编程习惯 2.5 避免标识符以下划线和双下划线开头，因为 Python 解释器可能保留了那些名称，供内部使用。这样可避免您选择的名称与解释器选择的名称混淆。

除了名称和值，每个对象还有一个“类型”。对象类型标识对象中存储的信息种类（比如整数、字符串等等），整数包括负整数（-14）、零（0）和正整数（6）。在诸如 C++ 和 Java 的语言中，程序员必须先声明对象类型，然后才能在程序中使用它。但是，Python 使用的是“动态类型定义”技术，即在程序执行期间决定一个对象的类型。例如，假如对象 `a` 初始化成 2，对象的类型会被定为 `integer`（整数），因为 2 是一个整数。类似地，如果对象 `b` 初始化成“Python”，对象的类型会成为 `string`（字符串）。函数 `raw_input` 返回类型为 `string` 的值，因此 `integer1` 引用的值属于 `string` 类型。

为了对 `integer1` 返回的值执行整数加法，必须先将字符串值转换成整数值。Python 的 `int` 函数（第 6 行）可将字符串或数字转换成整数值，并返回新值。如没有为变量 `integer1` 获得一个整数值，便无法获得希望的结果——程序会对两个字符串进行合并操作，而不是执行整数加法。图 2.8 对此进行了演示。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> value1 = raw_input("Enter an integer: ")
Enter an integer: 2
>>> value2 = raw_input("Enter an integer: ")
Enter an integer: 4
>>> print value1 + value2
24
```

图 2.8 事先不转换为整数便对 `raw_input` 返回的值进行相加，会得到错误结果（正确结果应该是 6）

赋值语句（图 2.7 的第 11 行）计算变量 `integer1` 和 `integer2` 的和，并将结果指派给变量 `sum`，这是用赋值符号=来完成的。这个语句可读作“`sum` 引用 `integer1 + integer2` 的值”。大多数计算都是通过赋值语句来执行的。

加号（+）是一个运算符，它是一个执行特定操作的特殊符号。本例中，运算符+执行加法运算，我们将其称为“二元运算符”，因为它需要两个操作数（值）才能执行运算。本例的两个操作数分别是 `integer1` 和 `integer2`。注意在 Python 中，符号=不是运算符而是赋值符。

常见编程错误 2.1 试图访问一个未赋值的变量，会产生运行时错误。

良好编程习惯 2.6 在二元运算符两端添加一个空格。这样可突出运算符，增强程序的可读性。

第 13 行显示字符串“Sum is”和变量 `sum` 的数值。我们想要输出的项目用逗号（,）分隔。注意，这个 `print` 语句输出的值具有不同的类型，即一个字符串和一个整数。

还可在输出语句中执行计算。可将第 11 行和第 13 行的语句合并成：

```
print "Sum is", integer1 + integer2
```


这可避免使用变量 `sum`。只有在认为合并能使程序更清晰的前提下，才可这样做。

2.5 内存概念

`integer1`, `integer2` 和 `sum` 等变量实际对应于 Python 的对象。每个对象都有自己的类型、大小、值以及在计算机内存中的位置。程序不可更改对象的类型或位置。有的对象类型允许程序员更改对象的值。第 5 章将详细地讨论这些类型。

图 2.7 的程序执行以下语句时：

```
integer1 = raw_input( "Enter first integer:\n" )
```

Python 首先创建一个对象，以容纳用户输入的字符串，然后将它放到一个内存位置。然后，赋值符`=`将名称 `integer1` 同新建的对象绑定（关联）起来。假定用户在 `raw_input` 提示处输入 45，Python 就会将字符串“45”放到与名称 `integer1` 对应的内存位置，如图 2.9 所示。

执行以下语句时：

```
integer1 = int( integer1 )
```

函数 `int` 会新建一个对象，用它来保存整数值 45。整数对象将从一个新内存位置开始，Python 将名称 `integer1` 同这个新内存位置绑定（图 2.10）。从此，变量 `integer1` 不再引用包含了字符串值“45”的内存位置。



图 2.9 显示一个变量值及其绑定名称的内存位置

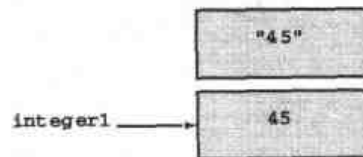


图 2.10 显示变量名称和值的内存位置

返回我们的求和程序，执行以下语句时：

```
integer2 = raw_input( "Enter second integer:\n" )
integer2 = int( integer2 )
```

假定用户输入字符串“72”。程序把这个值转换成整数值 72，并将其放到与 `integer2` 绑定的内存位置之后，内存的布局如图 2.11 所示。注意，这些对象的位置在内存中不一定是相邻的。

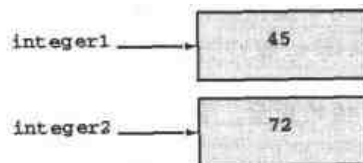


图 2.11 两个变量的值都已输入后的内存位置

程序获得 `integer1` 和 `integer2` 的值后，会把这些值加到一起并将结果指派给变量 `sum`。当以下语句：

```
sum = integer1 + integer2
```

执行加法后，内存布局如图 2.12 所示。注意在 `sum` 计算前后，`integer1` 和 `integer2` 的值是没有变化的。

换言之，从内存位置读取一个值时，该值不会改变。

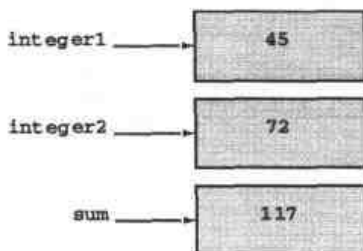


图 2.12 计算后的内存位置

从图 2.13 可以看出，每个 Python 对象都有一个位置、一个类型以及一个值，而且所有这些对象“属性”都是通过对象的名称来访问的。该程序和图 2.7 中的程序基本一致，只是我们在这一程序的各个地方，增添了一些语句以显示每个对象的内存位置、类型和值。

```

1 # Fig. 2.13: fig02_13.py
2 # Displaying an object's location, type and value.
3
4 # prompt the user for input
5 integer1 = raw_input("Enter first integer:\n") # read a string
6 print "integer1: ", id( integer1 ), type( integer1 ), integer1
7 integer1 = int( integer1 ) # convert the string to an integer
8 print "integer1: ", id( integer1 ), type( integer1 ), integer1
9
10 integer2 = raw_input("Enter second integer:\n") # read a string
11 print "integer2: ", id( integer2 ), type( integer2 ), integer2
12 integer2 = int( integer2 ) # convert the string to an integer
13 print "integer2: ", id( integer2 ), type( integer2 ), integer2
14
15 sum = integer1 + integer2 # assignment of sum
16 print "sum: ", id( sum ), type( sum ), sum
  
```

```

Enter first integer:
5
integer1: 7956744 <type 'str'> 5
integer1: 7637688 <type 'int'> 5
Enter second integer:
27
integer2: 7776368 <type 'str'> 27
integer2: 7637352 <type 'int'> 27
sum: 7637436 <type 'int'> 32
  
```

图 2.13 对象的位置、类型和值

完成 `raw_input` 调用后，第 6 行打印 `integer1` 的位置、类型和值。Python 函数 `id` 可返回解释器对变量内存位置的表示。函数 `type` 用于返回变量类型。将 `integer1` 的字符串值转换成整数值后，第 8 行再次打印这些值。注意在执行以下语句后，变量 `integer1` 的类型和位置都发生了变化：

```
integer1 = int( integer1 )
```

这一变化证明了程序不可更改变量类型这一事实。相反，这条语句会导致 Python 在新的内存位置创建一个新的整数值，并将 `integer1` 这个名称指派给该位置。`integer1` 以前引用的位置不能继续访问。示例程序剩余部分采用类似的方式，打印变量 `integer2` 和 `sum` 的位置、类型和值。

2.6 算术运算

大多数程序都要执行算术运算。图 2.14 总结了算术运算符。注意，一些特殊符号在标准代数中是没

有的。星号(*)代表乘法,百分号(%)是取模运算符,不久就会讨论这两个运算符。图 2.14 中的算术运算符是二元运算符,因为它们都要用到两个操作数。例如,表达式 `integer1 + integer2` 包含了二元运算符+以及两个操作数 `integer1` 和 `integer2`。

Python 运算	算术运算符	代数表达式	Python 表达式
加	+	$f + 7$	<code>f + 7</code>
减	-	$p - c$	<code>p - c</code>
乘	*	bm	<code>b * m</code>
求幂	**	x^y	<code>x ** y</code>
除	/	x/y 或 $\frac{x}{y}$ 或 $x \div y$	<code>x / y</code>
	// (Python 2.2 新增)		<code>x // y</code>
取模	%	$r \bmod s$	<code>r % s</code>

图 2.14 算术运算符

Python 是一种不断进步的语言,它的一些特性会随着时间的推移而发生变化。从 Python 2.2 起,除法运算符(/)的行为由“Floor 除法”变成“True 除法”。Floor 除法有时也称为整数除法,它将分子除以分母,返回不大于结果的最小的一个整数值。例如,如采用 Floor 除法,7 除以 4 结果是 1;17 除以 5 是 3。注意在 Floor 除法中,所有小数部分都被抛弃(即“截尾”),不进行进位处理。相反,True 除法产生精确的浮点结果,其中含有小数点,比如 7.0,0.0975 和 100.12345。例如,如采用 True 除法,7 除以 4 的结果就是 1.75。

在以前的版本中,Python 只提供一个除法运算符,即运算符/。运算符的行为(即 Floor 还是 True 除法)由操作数的类型决定。如操作数全是整数,就执行 Floor 除法。如一个或两个操作数是浮点数,就执行 True 除法。

语言设计者和许多程序员都不喜欢运算符/的这种随意性,所以决定在 2.2 版采用两个运算符。其中,运算符/执行 True 除法,运算符//则执行 Floor 除法。然而,在用老版本 Python 写的程序中,这样的设计有可能导致错误。因此,设计者采取了一个折衷方案:从 Python 2.2 开始,未来的所有 2.x 版本都会采用两个运算符,但假如程序作者希望使用新的行为,就必须使用以下语句明确表达自己的意愿:

```
from __future__ import division
```

Python 一旦看到这个语句,运算符/就会严格执行 True 除法,运算符//则严格/执行 Floor 除法。图 2.15 中的交互会话演示了这两种除法。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> 3 / 4      # floor division (default behavior)
0
>>> 3.0 / 4.0  # true division (floating-point operands)
0.75
>>> 3 // 4     # floor division (only behavior)
0
>>> 3.0 // 4.0 # floating-point floor division
0.0
>>> from __future__ import division
>>> 3 / 4      # true division (new behavior)
0.75
>>> 3.0 / 4.0  # true division (same as before)
0.75
```

图 2.15 运算符/行为的差异

首先对表达式 $3/4$ 进行求值。表达式的结果是值 0，因为运算符/对于两个整数操作数的默认行为是 Floor 除法。表达式 $3.0/4.0$ 的求值结果是 0.75。在这里，由于使用的是浮点操作数，所以运算符/会执行 True 除法。表达式 $3//4$ 和 $3.0//4.0$ 的求值结果分别为 0 和 0.0，因为运算符//总是执行 Floor 除法，不管操作数的对象是什么。然后，在第 13 行，我们用 import 语句更改运算符/的行为。实际上，该语句打开了运算符/的 True 除法能力。现在，表达式 $3/4$ 的计算结果是 0.75。注意本书只采用运算符/默认的 2.2 版本的行为，即：针对整数执行 Floor 除法（参见图 2.15 的第 5~6 行），针对浮点数执行 True 除法（参见图 2.15 的第 7~8 行）。

移植性提示 2.1 预计在 Python 3.0 中，运算符/只能执行 True 除法。3.0 版本发布后，程序员需要更新自己的程序，以兼容新行为。欲知详情，访问 python.sourceforge.net/peps/pep-0238.html。

Python 提供了取模运算符%，用于得到整除后的余数。 $x \% y$ 这个表达式会得到 x 被 y 除之后的余数。因此， $7 \% 4$ 等于 3，而 $17 \% 5$ 等于 2。这个运算符经常用于整数操作数，但也可用于其他算术类型。后文还会讨论取模运算符的许多有趣的应用，比如判断一个数字是否是另一个数字的倍数（它的一个特例便是判断一个数字是奇数还是偶数）。注意，取模运算符可用于整数和浮点数类型。

Python 中的算术表达式必须采用“直线”形式输入计算机。因此，像“ a 被 b 除”这样的表达式必须写成 a/b ，使所有常量、变量和运算符出现在一条直线中。下述代数表达式：

$$\frac{a}{b}$$

将是编译器或解释器无法接受的——尽管一些特殊用途的软件包允许以这种更自然的记号法来表示复杂的代数表达式。

在 Python 表达式中，圆括号的用法和代数表达式中大致相同。例如要将 a 乘以 $b+c$ 的和，可写成：

$$a * (b + c)$$

Python 根据由下述“运算符优先级规则”规定的精确顺序，在算术表达式中应用不同的运算符。这些规则同在代数中的规则大致相同：

1. 圆括号中的表达式先求值。所以程序员可根据自己的愿望，用圆括号强制按任何顺序求值。圆括号具有“最高优先级”。在嵌套或嵌入圆括号的情况下，由内向外求值。
2. 接着是求幂运算。如表达式包含几个求幂运算，就按从右到左的顺序进行。
3. 接着是乘法、除法和取模运算。如一个表达式中包含了几个乘、除和取模运算，就按从左到右的顺序求值。我们认为乘、除和取模具有相同的优先级。
4. 加、减运算最后进行。如果一个表达式同时包含了几个加法和减法运算，就按从左到右的顺序求值。加、减也具有相同的优先级。

如表达式中包含几对圆括号，这些圆括号并不一定是嵌套的。例如表达式：

$$a * (b + c) + c * (d + e)$$

并不包含嵌套圆括号。但该表达式中的圆括号具有相同的优先级。

当我们说特定的运算符从左到右求值时，事实上说的是运算符顺序的关联性。比如以下表达式中：

$$a + b - c$$

加法运算符(+)从左到右顺序关联。注意，也有一些运算符是从右到左顺序关联的。

图 2.16 总结了这些运算符优先级规则。介绍了更多的 Python 运算符后，我们还会对该表进行扩充。

运算符	运算	求值顺序
()	圆括号	最先求值。如果圆括号是嵌套的，最内层的表达式先求值。如果同时有几对圆括号具有相同优

运算符	运算	求值顺序
		优先级（即没有嵌套），就按从左到右的顺序求值
**	求幂	其次求值。如同时有几个，就按从右到左的顺序求值
* / // %	乘、除、取模	第三个求值。如同时有几个，就按从左到右的顺序求值。注意，运算符//是 Python 2.2 新增的
+或-	加、减	最后求值。如同时有几个，就按从左到右的顺序求值

图 2.16 算术运算符的优先级

现在，让我们基于运算符优先顺序来讨论几个例子。每个例子都列出了代数表达式及其对应的 Python 表达式。下面是求 5 个值的算术平均值的例子：

$$\text{代数: } m = \frac{a+b+c+d+e}{5}$$

Python: `m = (a + b + c + d + e) / 5`

圆括号是必需的，因为除法拥有的优先级比加法高。我们的目的是将 $(a+b+c+d+e)$ 除以 5。遗漏了圆括号，得到的就是 $a+b+c+d+e/5$ 。它等价于下面这个不正确的代数表达式：

$$a+b+c+d+\frac{e}{5}$$

下面是一个直线形式的方程式例子：

代数: $y = mx + b$

Python: `y = m * x + b`

无需任何圆括号。首先会执行乘法，因为乘法具有比加法更高的优先级。

下例包含了取模 (%)、乘法、除法、加法和减法运算：

代数: $z = pr\%q + w/x - y$

Python: `z = p * r % q + w / x - y`

① ② ④ ③ ⑤

圆圈内的数字指明 Python 执行运算的顺序。首先，乘法、取模和除法运算符按从左到右的顺序依次执行（即从左到右顺序关联），这是由于它们的优先级要高于加法和减法。接着执行的是加法和减法运算符——也是按从左到右的顺序。一旦表达式求值结束，Python 就将结果指派给变量 `z`。

为了更好地理解运算符优先级规则，我们来看看一个二次多项式是如何求值的：

`y = a * x ** 2 + b * x + c`

⑥ ② ① ④ ③ ⑤

圆圈内的数字指明 Python 执行运算时的顺序。

假定变量 `a`, `b`, `c` 和 `x` 像下面这样初始化：`a = 2`, `b = 3`, `c = 7` 和 `x = 5`，就可用图 2.17 来展示在上述二次多项式中运算符的应用顺序。

以上赋值语句也可用可有可无的圆括号来强调语句的清晰性，如下所示：

`y = (a * (x ** 2)) + (b * x) + c`

良好编程习惯 2.7 和在代数中一样，可在表达式中添加原本不需要的圆括号，使其更清晰。这些括号叫做冗余括号。冗余括号通常用于分组大型表达式中的各个子表达式，使表达式更清晰。将一条长的语句分解成一系列较短的、较简单的语句，有助于使语句更清晰。

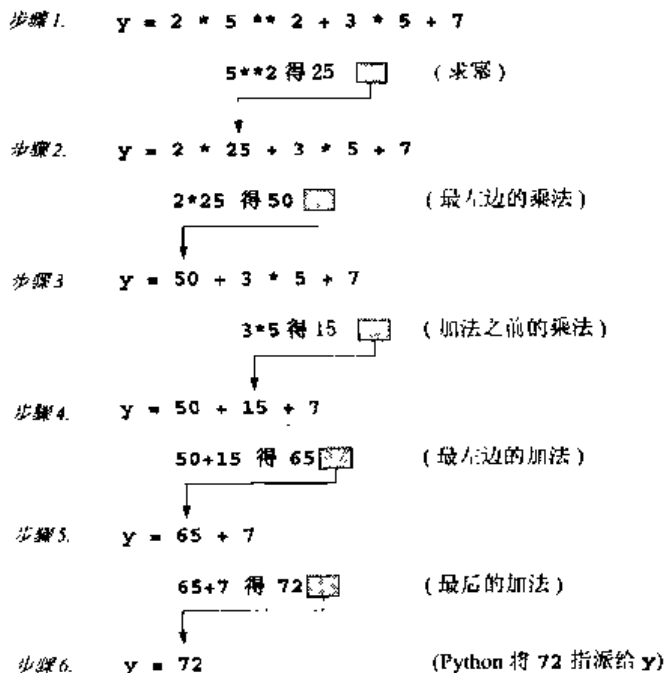


图 2.17 二次多项式求值顺序

2.7 字符串格式化

前面讨论了数值的问题，现在来看看字符串。和其他流行的程序语言不同，Python 将字符串作为内建数据类型提供，这使得 Python 程序可执行非常强大的、基于文本的处理。我们已学习了如何将文本放在双引号内 (") 以创建一个字符串。Python 字符串还可采用其他许多方式创建，如图 2.18 所示。

```
1 # Fig. 2.18: fig02_18.py
2 # Creating strings and using quote characters in strings.
3
4 print "This is a string with \"double quotes.\"\"
5 print 'This is another string with "double quotes."'
6 print 'This is a string with \'single quotes.\''
7 print "This is another string with 'single quotes.'"
8 print """This string has "double quotes" and 'single quotes'.
9     You can even do multiple lines."""
10 print '''This string also has "double" and 'single' quotes.'''
```

```
This is a string with "double quotes."
This is another string with "double quotes."
This is a string with 'single quotes.'
This is another string with 'single quotes.'
This string has "double quotes" and 'single quotes'.
    You can even do multiple lines.
This string also has "double" and 'single' quotes.
```

图 2.18 创建 Python 字符串

第 4 行用大家熟悉的双引号字符来创建一个字符串。如果希望在屏幕上打印双引号，必须使用双引号字符的转义序列，即 \，不能只使用一个双引号。

字符串还可单引号 (') 创建，如第 5 行所示。如希望在用单引号创建的字符串中使用双引号字符，就不需要转义字符。类似地，要想在用双引号创建的字符串中使用单引号字符，也不必使用转义字符 (第 7 行)。然而，如果想在用单引号创建的字符串中使用单引号字符 (第 6 行)，就必须使用转义序列 \。

Python 还支持三引号字符串，如第 8~10 行所示。三引号字符串尤其适合在字符串中输出特殊字符，

比如引号字符。在三引号字符串中，单或双引号字符不必添加转义字符。另外，也可将三引号字符用于大型的文本块，因为三引号字符串可跨越多行。本书中，示例程序要为 Web 输出大的文本块时，就会使用三引号字符串。

Python 字符串支持简单但功能强大的输出格式化。可采用几种方式对输出的字符串进行格式化：

1. 为浮点值取指定的小数位。
2. 使用指数（科学）计数法表示浮点数。
3. 使一系列数字的小数点对齐。
4. 使输出右对齐或左对齐。
5. 在一行输出的指定位置插入字符或字符串。
6. 用固定字段宽度和精度显示所有类型的数据。

图 2.19 的程序演示了基本的字符串格式化功能。第 4~7 行演示了如何在字符串中表示整数。第 5 行直接显示变量 `integerValue` 的值，不进行任何格式化。格式化运算符 `%` 可在字符串中插入一个变量值（第 6 行）。运算符左边的值是一个字符串，它包含一个或多个“转换指示符”（值在字符串中的占位符）。每个转换指示符都以百分号（`%`）开头——不要把它同格式化运算符 `%` 弄混了——并以一个“转换指示符符号”结尾。转换指示符符号 `d` 表明我们希望在当前字符串的指定位置放入一个整数。图 2.20 总结了用于字符串格式化的几个转换指示符符号。

格式化运算符 `%` 的右边，指定了要用来替换字符串中的占位符。在第 6 行，我们指定要用值 `integerValue` 来替换字符串中的占位符 `%d`。第 7 行将变量 `integerValue` 的值转换成十六进制形式后插入字符串。

```
1 # Fig. 2.19: fig02_19.py
2 # String formatting.
3
4 integerValue = 4237
5 print "Integer ", integerValue
6 print "Decimal integer %d" % integerValue
7 print "Hexadecimal integer %x\n" % integerValue
8
9 floatValue = 123456.789
10 print "Float", floatValue
11 print "Default float %f" % floatValue
12 print "Default exponential %e\n" % floatValue
13
14 print "Right justify integer (%8d)" % integerValue
15 print "Left justify integer (%-8d)\n" % integerValue
16
17 stringValue = "String formatting"
18 print "Force eight digits in integer %8d" % integerValue
19 print "Five digits after decimal in float %.5f" % floatValue
20 print "Fifteen and five characters allowed in string:"
21 print "(%.15s) (%.5s)" % (stringValue, stringValue)
```

```
Integer 4237
Decimal integer 4237
Hexadecimal integer 108d

Float 123456.789
Default float 123456.789000
Default exponential 1.234568e+005

Right justify integer ( 4237)
Left justify integer (4237 )

Force eight digits in integer 00004237
Five digits after decimal in float 123456.78900
Fifteen and five characters allowed in string:
(String formatti) (Strin)
```

图 2.19 字符串格式化运算符 `%`

转换指示符号	含义
c	单个字符（即长度为 1 的字符串），或者一个 ASCII 字符的整数表示
s	字符串，或者要转换成字符串的一个值
d	有符号（正负号）的整数
u	无符号十进制整数
o	无符号八进制整数
x	无符号十六进制整数（a 到 f 的数位采取小写形式）
X	无符号十六进制整数（A 到 F 的数位采取小写形式）
f	浮点数
e, E	浮点数（使用科学计数法）
g, G	浮点数（采用最低有效数位）

图 2.20 字符串格式化字符

第 9~12 行演示了如何在字符串中插入浮点值。转换指示符充当一个浮点值的占位符（第 11 行）。在格式化运算符%的右边，将变量 floatValue 用作要显示的值。e 转换指示符是一个采用指数计数法的浮点值的占位符。指数计数法等价于数学中的科学计数法，只是前者在计算机中使用。例如，150.4582 可采用科学计数法表示为 1.504582×10^2 ，计算机的指数计数法则将其表示成 1.504582E+002，即 1.504582 乘以 10 的 2 次方（E+002）。E 是 Exponent（指数）的简称。

第 14~15 行演示了如何用“字段宽度”来格式化字符串。字段宽度是指一个字段的的最小长度，值将在这个字段中打印。如字段宽度大于要打印的值的宽度，数据通常在字段内“右对齐”。要使用字段宽度，必须在百分号和转换指示符之间插入一个整数以指定字段宽度。第 14 行在宽度为 8 的一个字段中右对齐显示变量 integerValue 的值。要想左对齐，需要将字段宽度设为负数（第 15 行）。

第 17~21 行演示了如何用精度来格式化字符串。对于不同数据类型，“精度”有不同的含义。如果用于整数转换指示符，精度指定的是要打印的最小位数。如打印值包含的位数少于指定的精度，就为打印的值加上前置零，直到总位数等于精度。要想使用精度，请在百分号和转换指示符符号之间插入一个小数点（.），并后跟一个代表精度的整数。例如，第 18 行在打印变量 integerValue 的值时，采用了 8 位精度。

将精度用于一个浮点转换指示符时，精度就是小数点后出现的位数。第 19 行在打印变量 floatValue 时，采用了 5 位精度。

用于字符串转换指示符时，精度是指从字符串中提取并显示的最大字符数。第 21 行打印两次变量 stringValue 的值，一次精度为 15，另一次为 5。注意，转换指示符包含在圆括号中。若格式化运算符%左边的字符串包含多个转换指示符，右边的值就必须是一个用逗号分隔的值序列。该序列包含在圆括号中，而且值的数量必须与含有转换指示符的字符串的数量一致。Python 按从左到右的顺序构造字符串——用圆括号中的值逐个替换格式化字符。

Python 字符串还通过“字符串方法”，支持更强大的字符串格式功能，详情参见第 13 章。

2.8 做出决策：相等运算符和关系运算符

针对 Python 的 if 结构，本节要介绍一个简单版本，它允许程序根据一些条件是否成立，从而做出决策。如条件成立（条件为真），if 结构主体中的语句就会执行。如条件不成立（条件为假），主体语句就不执行。稍后会介绍一个例子。

if 结构中的条件可用如图 2.21 所示的相等运算符和关系运算符构建。关系运算符全都在同一个优先级上，并从左到右顺序关联。所有相等运算符也在同一个优先级上，但其优先级低于关系运算符。相等运算符也从左到右顺序关联。

常见编程错误 2.2 ==、!=、>=和<=这几个运算符的两个符号之间出现空格，会造成语法错误。

常见编程错误 2.3 !=, <>, >=和<=这几个运算符中，假如两个字符的顺序弄反了（分别写成!=, ><, =>和=<），会造成语法错误。

常见编程错误 2.4 切不可将相等运算符“==”同赋值运算符“=”弄混了。其实按正统逻辑，在读的时候，相等运算符才应读成“...等于...”。赋值运算符则应该读成“...获得...”、“获得...的值”或者“被赋值为...”。有人喜欢把相等运算符读成“等于等于”。在 Python 中，如果在条件语句中使用了错误的赋值符号，会造成语法错误。

标准的代数相等运算符 或关系运算符	Python 相等运算符或 关系运算符	Python 条件示例	Python 条件 的含义
关系运算符			
>	>	x > y	x 大于 y
<	<	x < y	x 小于 y
≥	>=	x >= y	x 大于或等于 y
≤	<=	x <= y	x 小于或等于 y
相等运算符			
=	==	x == y	x 等于 y
≠	!=, <>	x != y x <> y	x 不等于 y

图 2.21 相等运算符和关系运算符

下例用 6 个 if 结构来比较两个用户输入的数字。满足任何一个 if 条件，就会执行与 if 对应的 print 语句。用户要输入两个值，程序将其转换成整数，并指派给变量 number1 和 number2。然后，程序比较这两个数字并显示比较结果。图 2.22 展示了示例程序以及一些示范结果。

```

1 # Fig. 2.22: fig02_22.py
2 # Compare integers using if structures, relational operators
3 # and equality operators.
4
5 print "Enter two integers, and I will tell you"
6 print "the relationships they satisfy."
7
8 # read first string and convert to integer
9 number1 = raw_input( "Please enter first integer: " )
10 number1 = int( number1 )
11
12 # read second string and convert to integer
13 number2 = raw_input( "Please enter second integer: " )
14 number2 = int( number2 )
15
16 if number1 == number2:
17     print "%d is equal to %d" % ( number1, number2 )
18
19 if number1 != number2:
20     print "%d is not equal to %d" % ( number1, number2 )
21
22 if number1 < number2:
23     print "%d is less than %d" % ( number1, number2 )
24
25 if number1 > number2:
26     print "%d is greater than %d" % ( number1, number2 )
27
28 if number1 <= number2:

```

```

29     print "%d is less than or equal to %d" % ( number1, number2 )
30
31 if number1 >= number2:
32     print "%d is greater than or equal to %d" % ( number1, number2 )

```

```

Enter two integers, and I will tell you
the relationships they satisfy.
Please enter first integer: 37
Please enter second integer: 42
37 is not equal to 42
37 is less than 42
37 is less than or equal to 42

```

```

Enter two integers, and I will tell you
the relationships they satisfy.
Please enter first integer: 7
Please enter second integer: 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7

```

```

Enter two integers, and I will tell you
the relationships they satisfy.
Please enter first integer: 54
Please enter second integer: 17
54 is not equal to 17
54 is greater than 17
54 is greater than or equal to 17

```

图 2.22 用相等运算符和关系运算符判断逻辑关系

程序使用 Python 函数 `raw_input` 和 `int` 输入两个整数（第 8~14 行）。首先获得变量 `number1` 的值，然后获得 `number2` 的值。

第 16~17 行的 `if` 结构比较变量 `number1` 和 `number2` 的值，测试其相等性。如果两个值相等，就显示一行文本，指出数字是相等的（第 17 行）。如果满足一个或多个从第 19 行、第 22 行、第 25 行、第 28 行和第 31 行开始的 `if` 结构的条件，相应的 `print` 语句就会显示一行文本。

每个 `if` 结构都包括单词 `if`、要测试的条件以及一个冒号（:）。`if` 结构还包含一个主体（称为 `suite`）。图 2.22 中的每个 `if` 结构主体都只包含一个语句，而且每个主体都是缩进的。一些语言（如 C++、Java 和 C#）用花括号 `{}` 标明 `if` 结构主体；Python 则用“缩进”来达到这个目的。下一节会详细讨论 Python 的缩进问题。

常见编程错误 2.5 忘记在 `if` 结构中插入冒号（:）是语法错误。

常见编程错误 2.6 忘记对 `if` 结构的主体进行缩进（缩排）是语法错误。

良好编程习惯 2.8 事先建立一个约定，设置您喜欢的缩进量，然后始终贯彻这一约定。虽然按 `Tab` 键可生成缩进，但制表位的长度在不同系统上是不同的。建议将 3 个空格定为一个缩进级别。

Python 对语法的评估取决于空距；因此，如果空距的用法不统一，可能会造成语法错误。例如，将语句分解为多行可能导致语法错误。如语句过长，可使用续行字符 `\` 将其分为多行。有的 Python 解释器用 `...` 来注明一个延续的行。图 2.23 中的交互式会话演示了续行字符的用法。

```

Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print 1 +
      File "<string>", line 1
        print 1 +
              ^
SyntaxError: invalid syntax
>>> print 1 + \

```

```
... 2
3
>>>
```

图 2.23 续行字符 (\)

良好编程习惯 2.9 可用续行字符\将长语句分为几行。如一个语句必须分解成多行，请选择有意义的断点，比如在 print 语句的逗号之后，或者在一个较长表达式的运算符之后。

图 2.24 总结了本章介绍的运算符的优先级。各运算符根据优先级，从上到下降序排列。注意，除了求幂运算运算符之外，其他所有运算符都从左到右顺序关联。

运算符	顺序关联性	类型
()	从左到右	圆括号
**	从右到左	求幂
* / // %	从左到右	乘
+ -	从左到右	加
< <= > >=	从左到右	关系
== != <>	从左到右	相等

图 2.24 已讨论过的运算符的优先级和顺序关联性

测试和调试提示 2.3 如果一个表达式里包含许多运算符，请务必参考运算符优先级表，核实表达式中的运算符按自己希望的顺序执行。如表达式过于复杂以至于无法确定顺序，不妨将表达式分割为几个小语句，或干脆用圆括号强行规定顺序——在代数表达式中也可采用一样的做法。注意，某些运算符（比如求幂运算符**）是按右到左顺序关联的，而非从左到右。

2.9 缩进

Python 利用代码文本的缩进来定界（区分）代码的不同区域。其他程序语言则通常使用花括号来定界不同区域。在 Python 中，一个“suite”是指同控制结构的主体对应的一个代码区域。第 3 章将介绍“block”（块或代码块）的概念，它通常就是函数的主体。Python 程序员要自行决定一个 suite 或 block 的缩进量。另外，suite 或 block 中每个语句的缩进量都必须保持一致。Python 通过缩进量的变化来识别新的 suite 或 block。

常见编程错误 2.7 如果一个区域包含的代码行没有统一进行缩进，Python 解释器会认为那些行从属于其他区域，因而造成语法或逻辑错误。

图 2.25 对图 2.22 中的程序进行了修改，演示了不正确缩进的后果。第 21~22 行显示的是 if 结构的错误缩进。尽管程序不会报错，但它确实跳过了一个相等运算符。第 21 行的语句：

```
if number != number:
```

只有在第 16 行的 if number1 == number2 语句执行的前提下才会执行。本例中，第 21 行的 if 语句永远不会执行，因为两个相等的数字肯定不会不相等（2 不可能不等于 2）。所以，图 2.25 中的输出本应指出 1 is not equal to 2（1 不等于 2），但因上述错误无法显示。

```
1 # Fig. 2.25: fig02_25.py
2 # Using if statements, relational operators and equality
3 # operators to show improper indentation.
4
5 print "Enter two integers, and I will tell you"
6 print "the relationships they satisfy."
```

```

7
8 # read first string and convert to integer
9 number1 = raw_input( "Please enter first integer: " )
10 number1 = int( number1 )
11
12 # read second string and convert to integer
13 number2 = raw_input( "Please enter second integer: " )
14 number2 = int( number2 )
15
16 if number1 == number2:
17     print "%d is equal to %d" % ( number1, number2 )
18
19     # improper indentation causes this if statement to execute only
20     # when the above if statement executes
21     if number1 != number2:
22         print "%d is not equal to %d" % ( number1, number2 )
23
24 if number1 < number2:
25     print "%d is less than %d" % ( number1, number2 )
26
27 if number1 > number2:
28     print "%d is greater than %d" % ( number1, number2 )
29
30 if number1 <= number2:
31     print "%d is less than or equal to %d" % ( number1, number2 )
32
33 if number1 >= number2:
34     print "%d is greater than or equal to %d" % ( number1, number2 )

```

```

Enter two integers, and I will tell you
the relationships they satisfy.
Please enter first integer: 1
Please enter second integer: 2
1 is less than 2
1 is less than or equal to 2

```

图 2.25 用于演示错误缩进的 if 语句

测试和调试提示 2.4 为避免难以察觉的错误，务必在 Python 程序中采用统一和正确的缩进。

2.10 对象思想：对象技术简介

本书前 6 章的重点是结构化编程的“常规”方法论，这是因为即将构建的对象会包含一部分结构化程序组件。本节提前介绍面向对象的概念。通过本节的学习，您能理解面向对象是一种思考这个世界和编写计算机程序的一种自然方式。

首先从一些关键概念和术语开始“面向对象之旅”。观察一下身旁的现实世界，对象无处不在——人、动物、植物、汽车、飞机、建筑物、计算机等等，一切都是对象。人们通过对象来思考问题。每个人都有非凡的抽象能力，可将屏幕上显示的一幅图像看成人、飞机、树和山这样的对象，而非看成单独的颜色点。如果愿意，我们可以看到沙滩而不是沙子，看到森林而不是树木，看到房子而不是砖块。

人们习惯将对象划分为两大类——动物和静物。其中，动物在某种程度上是“有生命的”。它们四处运动，并做一些事情。静物，比如毛巾等，则表面上什么事情也不做。它们只是某种静态的物品。然而，所有这些对象都有一些东西是共通的。它们都有自己的一些属性，例如大小、形状、颜色、重量等等；而且都在表现出某种行为，例如球会滚动、弹跳、膨胀和收缩，婴儿会哭、睡、爬、走和眨眼睛，车会加速、刹车和转弯，毛巾会吸水等等。

人们通过研究其属性和观察其行为，从而了解对象。不同的对象可能具有相同的属性，并可表现出类似的行为。例如，可以比较一下婴儿和成人，人和猩猩等等。对于轿车、卡车、小童车和溜冰鞋等等来说，它们其实也有许多共通之处。

面向对象编程（OOP）以软件形式对现实世界对象进行建模。它利用了类关系；在这种关系中，特

定类（比如一个汽车类）的所有对象都具有相同的特征。它还利用了继承关系，甚至利用了多重继承关系。在这种关系中，通过吸收现有类的特征建立新的对象类，同时可能添加了自己独有的一些特征。“敞篷车”类的一个对象肯定具有一个更常规的“汽车”类的特征，但敞篷车的车篷还可展开和折叠，这是普通汽车不具备的。

面向对象的编程还为我们提供了一种更加自然和直观的方式来看待编程过程。换言之，编程过程就是对现实对象及其属性/行为进行建模的一种过程。OOP 还可为对象之间的通信进行建模。就和人们向同伴发送消息一样（例如军官命令士兵立正），对象之间也通过消息来沟通。

OOP 将数据（属性）和函数（行为）封装到一种名为“对象”的包中；一个对象的数据同函数紧密联系在一起。对象具有“信息隐藏”的特点。也就是说，尽管一个对象可能知道如何通过经良好定义的接口与另一个对象通信，但一个对象通常不允许知道其他对象内部是如何实现的——实现的细节被隐藏在对象内部。这是非常合理的，我们完全能很好地驾驶一辆汽车，而不必了解发动机、传动机构和排气系统的内部工作原理。以后，我们会解释信息隐藏对于保障“良好的软件工程”为何如此重要。

在 C 和其他过程式程序语言中，编程通常是“面向行动”的；但在 Python 中，编程是“面向对象”的（在理想情况下）。在过程式语言中，基本编程单元是“函数”；而在面向对象语言中，基本编程单元是“类”，最终通过它实例化（即“创建”的一种更好听的说法）出对象。Python 类包含了函数（用于实现类的行为）和数据（用于实现类的属性）。

采用过程式编程，程序员可把重点放在写函数上。用于执行一些任务的行动被组合成函数，不同的函数进一步组合，即构成程序。在过程式编程中，数据肯定是重要的，但数据的主要用途是为函数执行的行动提供支持。在一份系统规格说明（描述应用程序所提供的服务的一份文档）中，那些动词帮助过程式程序员确定并设计一系列协同工作的函数，它们用于实现整个系统。

采用面向对象编程，程序员则把重点放在创建他们自己的用户自定义类型上，即“类”。每个类都包含数据以及一系列函数，函数用于对数据进行处理。一个类的数据组件称为数据成员或属性；一个类的函数组件则称为方法（在其他面向对象编程语言中，也可能称为“数据成员”）。面向对象编程侧重于类，而不是函数。在一份系统规格说明中，那些名词帮助面向对象程序员确定并设计一系列类，以便通过它们创建一系列协同工作的对象，这些对象用于实现整个系统。

类之于对象，犹如蓝图之于房屋。可依据蓝图建造许多房屋，也可依据一个类创建许多对象。一个类可与其他类建立关系。例如，假定一家银行采取了面向对象的设计，那么 BankTeller（出纳员）类需要同 Customer（客户）类建立关系。这种关系称为“关联”。

我们以后会体验到，一旦软件被打包成类，这些类便可重复用于未来的软件系统。一组相关的类通常打包成可重复使用的“组件”或“模块”。房地产经纪人会告诉他们的客户，影响房地产价格的三大因素是“地段，地段，还是地段”。类似地，我们也认为影响软件开发前途的三大因素是“重用，重用，还是重用”。

事实上，利用对象技术，今后大多数软件都可通过合成一系列“标准化、可互换零件”而构建，这些零件称为“组件”。本书将教您如何“创建宝贵的类”，以达到重用、重用、再重用之目的。您创建的每个新类都可能成为一项价值无限的软件资产，您和其他程序员可用它加快未来软件开发的速度，并改进其质量。类的重用，大有潜力可挖。

本章介绍了 Python 的大量重要特性，包括在屏幕上打印数据，从键盘输入数据，执行计算和做出决策等。第 3 章介绍“结构化编程”时，会频繁用到这些技术。届时，您将学习如何指定和更改语句执行顺序——该顺序称为“控制流程”。本章还介绍了面向对象的基本概念和术语。第 7~9 章将继续讨论面向对象编程。

第3章 控制结构

学习目标

- 理解基本的求解方法
- 会利用“自上而下求精法”开发自己的算法
- 会使用 if、if/else 和 if/elif/else 结构选择恰当的行动
- 会用 while 和 for 重复结构在一个程序中重复执行语句
- 理解由计数器控制的重复，以及由哨兵值控制的重复
- 会使用增量赋值符号和逻辑运算符
- 会使用 break 和 continue 程序控制语句

3.1 概述

为解决特定的问题而写一个程序之前，首先有必要透彻理解该问题，并精心计划好每一步操作，最终圆满解决问题。写程序时，理解基本构建单元的类型，并严格遵守已证明行之有效的程序构建原则，同样不可忽视。本章将讨论所有这些问题，向大家展示结构化编程的理论及原则。注意，这里描述的技术适用于大多数高级程序语言。从第7章开始学习面向对象编程时，我们会利用本章介绍的控制结构来构建和操纵对象。

3.2 算法

任何计算问题都可通过按指定顺序采取一系列行动得到解决。所谓“算法”，是指解决一个问题的“过程”，它有两方面的含义：

1. 要采取的“行动”
2. 采取这些“行动”的顺序

下例演示了正确指定行动顺序的重要性。

假定要设计一个“起床上班算法”，模拟一名职员早上起床并去上班的过程。他的行动是：① 起床；② 脱下睡衣；③ 洗个淋浴；④ 穿好衣服；⑤ 吃早餐；⑥ 开车上班。只有按以上顺序行动，职员才能获得最高效率。

如果调换一下顺序，变成：① 起床；② 脱下睡衣；③ 穿好衣服；④ 洗个淋浴；⑤ 吃早餐；⑥ 开车上班，可怜的职员只好穿着湿淋淋的衣服去上班了。

在计算机程序中，指定语句执行顺序的操作称为“程序控制”。本章将探讨 Python 的程序控制。

3.3 伪代码

伪代码（Pseudocode）是一种虚的、非正式的语言，目的是帮助程序员设计算法。伪代码包括对“可执行语句”的描述——一旦从伪代码转换成 Python 语句，并开始运行，这些语句就可以执行。利用这里提供的伪代码，可开发出实际、有效的算法，并可方便地转换成 Python 程序。伪代码就像我们的日常语言；既方便，又好用——尽管它并不是真正的计算机程序语言。

伪代码程序无法在计算机上执行。它们惟一的用处是，在使用 Python 等程序语言编写正式代码之前，

可帮助程序员“计划”好整个程序。本章用几个例子演示了如何通过伪代码高效率地开发 Python 程序。

软件工程知识 3.1 程序设计过程中，常用伪代码来“思考”一个程序，再将伪代码程序转换成 Python 程序。

我们展示的伪代码程序纯粹由字符构成，所以程序员可用一个编辑器程序方便地输入这些代码。计算机可根据需要显示伪代码程序的最新拷贝。对精心设计的伪代码程序来讲，它们可轻松转换成相应的 Python 程序。许多时候，都只需用对应的 Python 语句替换伪代码语句。

3.4 控制结构

通常，程序中的语句是按原先书写的顺序执行的。这称为“顺序执行”。然而，许多 Python 语句允许程序员自行控制接着要执行的语句，该语句可能是、也可能不是顺序的“下一条”语句。这称为“转交控制权”，具体由 Python 的“控制结构”来实现。本节要讨论控制结构开发的背景知识，并介绍 Python 用于在程序中转交控制权的特定工具。

20 世纪 60 年代，大量事实证明，假如胡乱转交控制权，会使软件开发变成一件令人非常头疼的事情。于是，人们将此归咎于 goto 语句（在包括 C 和 Basic 在内的几种程序语言中使用）。利用该语句，程序员可将控制权转交给一个程序里的几乎任意位置。因此，结构化编程的概念问世时，就郑重宣布自己是“免 goto”的。

Bohm 和 Jacopini 的研究证实，一个程序完全可以不必使用任何 goto 语句。大势所趋下，程序员们也逐渐改变了自己对 goto 的“偏爱”，变成“较少 goto 编程”。直到 20 世纪 70 年代，程序员们才开始认真看待“结构化编程”。结果令人振奋。采用结构化编程后，程序员们普遍的反映是，一个软件项目的各个方面（无论开发时间、速度还是成本），都得到了有效改善。之所以取得这些成功，是由于结构化程序条理更清晰、更易调试、更易修改，而且一开始就能避免大多数编程错误。

Bohm 和 Jacopini 的研究结果表明，事实上只需 3 种控制结构便可写出所有程序。这 3 种控制结构包括：顺序结构、选择结构以及重复结构。其中，顺序结构是 Python 内建的。除非特别声明，否则计算机都会顺序执行 Python 语句。图 3.1 的流程图演示了一个典型的顺序结构，两个计算将依序执行。“流程图”（Flowchart）是对全部或部分算法的一种图形化表示。

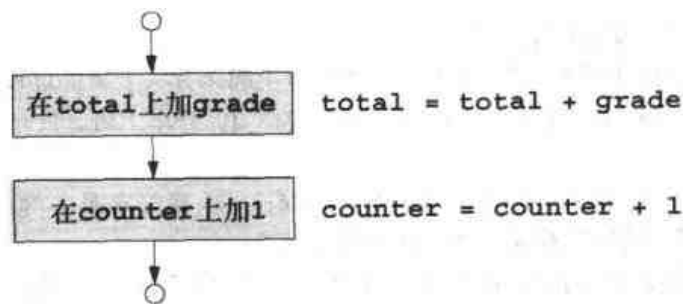


图 3.1 顺序结构流程图

流程图往往会使用一些具有特殊意义的符号，比如矩形、菱形、椭圆以及小圆圈等等。这些符号通过一些箭头线（称为流线，即 Flowline）连接起来，它指明算法采取行动的顺序。类似于伪代码，流程图在开发和表示算法时特别有用。尽管大多数程序员最喜欢的还是伪代码，但流程图明确指出了控制结构的运作机制。读者应仔细比较每种控制结构的伪代码及流程图表示方法。

请观察图 3.1 的顺序结构流程图，它用矩形符号（或称行动符号）指出每种类型的行动，包括计算或输入/输出操作等。图中的流线指出行动的执行顺序。首先，将 grade 加到 total 上，再把 1 加到 counter 上。Python 允许在顺序结构中采取数量不限的行动。在可以放入单个行动的任何位置，都可顺序放入几个行动。

要用流程图表示一个完整的算法，应将椭圆符号作为流程图的第一个符号，并在其中写上“开始”；最后一个符号也是一个椭圆符号，并在其中写上“结束”。但如图 3.1 所示，如果绘制的只是算法的一部分，头尾的椭圆符号都可省略，分别换成一个小圆圈符号（或称接头符号）。

在所有流程图符号中，最重要的或许是菱形符号（或称决策符号），它指出要在那个位置做出一项决定。下一节将更深入地讨论菱形符号。这里展示的伪代码有助于开发出能直接转换成结构化 Python 程序的算法。

Python 提供 3 种类型的选择结构：if、if/else 和 if/elif/else，本章要分别讨论它们。假如某个条件成立（条件为 true），if 选择结构会执行（选择）一项行动；假如不成立（条件为 false），则跳过这项行动。如条件成立（true），if/else 选择结构会采取一项行动；条件不成立（false），则采取另一项行动。if/elif/else 选择结构则在多个不同的行动中选择一个，具体取决于几个条件成立还是不成立。

if 选择结构是一种单选结构——要么选择、要么忽略一项行动；if/else 是一种双选结构——在两项不同的行动中做出选择；if/elif/else 则是一种多选结构——在许多不同的行动中选择实际要执行的。

Python 提供 2 种类型的重复结构：while 和 for。注意，if、else、switch、while、do 和 for 都是 Python 关键字。这些关键字是该语言为自己保留的，用于实现控制结构这样的 Python 特性。千万不要将关键字用作标识符（比如变量名）。图 3.2 总结了所有 Python 关键字。^①

Python 关键字						
and	continue	else	for	import	not	raise
assert	def	except	from	in	or	return
break	del	exec	global	is	pass	try
class	elif	finally	if	lambda	print	while

图 3.2 Python 关键字

常见编程错误 3.1 将关键字用作标识符是语法错误。

单入/单出控制结构便于构建程序。只需将一个控制结构的出口同下一个控制结构的入口连接到一起，便可将不同控制结构有机联系到一起。这类似于小孩子搭积木，所以，我们把这种方法称为“控制结构堆叠”（Control-Structure Stacking）。另外还有一种方法叫做“控制结构嵌套”（Control-Structure Nesting），它也能连接控制结构，稍后还会对此详加讨论。

软件工程知识 3.2 所有 Python 程序都可基于前述 6 类控制结构来创建（顺序、if、if/else、if/elif/else、while 和 for）。不同控制结构可采用 2 种方法连接，即控制结构堆叠和嵌套。

3.5 if 选择结构

利用选择结构，可在大量候选行动中挑选一个。例如，假定通过一次考试的分数线是 60 分，那么伪代码应该像下面这样：

```
假如 (If) 学生成绩大于或等于 60
    打印 "Passed"
```

在上述代码中，它判断的是“学生成绩大于或等于 60”这个条件为 true（真）还是为 false（假）。如条件为 true，则打印一条“Passed”，再顺序“执行”下一条伪代码语句（记住伪代码并不是真正的程序语言）。如条件为 false，则忽略打印语句，并顺序执行下一条伪代码语句。选择结构第 2 行进行了缩进处理。虽然并不一定要缩进（对伪代码而言），但建议您这么做，因为它强调了结构化程序固有的层次结构。

^① 请访问 Python 网站（www.python.org），查看这些关键字的一个列表，并注意避免将它们用作标识符。

将伪代码转换成 Python 代码时，则必须进行缩进。

在 Python 中，上述伪代码 *if* 语句可写成：

```
if grade >= 60:
    print "Passed"
```

注意 Python 代码同伪代码是严格对应的。正是因为具有这种关系，伪代码才能成为强有力的一种开发工具。*if* 结构主体中的语句输出字符串 "Passed"。

图 3.3 的流程图向大家揭示了单选 *if* 结构和菱形（决策）符号。该符号包含一个表达式（比如条件），它要么为 *true*，要么为 *false*。从菱形符号必须引出两条流线：一条指出表达式为 *true* 时的程序执行方向，另一条指出表达式为 *false* 时的程序执行方向。通过第 2 章的学习，您知道可根据包含关系或相等运算符的条件做出决策。实际上，一项决策可基于任何表达式——如表达式的值为零，就把它当作 *false*；如表达式的值不是零（非零），就把它当作 *true*。

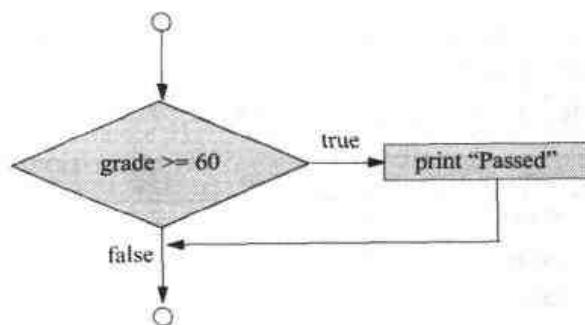


图 3.3 if 单选结构流程图

注意，*if* 结构是“单入/单出”结构。在其他控制结构流程图中，除小圆圈和流线外，还可包含矩形和菱形——前者指出行动，后者指出决策。这种类型的流程图强调了“行动 / 决策编程模型”。

想象有 6 个柜子，每个都包含了 6 种控制结构中的一种。这些控制结构是空的。在矩形和菱形符号中，均不包含任何内容。然后，程序员的任务是在惟一可选的两种方法中挑选一种（堆叠或嵌套），将这些控制结构合并到一起。然后，依据计划好的算法，用适当方式在其中填充行动和决策。稍后会讲解编写行动及决策内容的各种方式。

3.6 if/else 和 if/elif/else 选择结构

只有条件为 *true*，*if* 选择结构才会执行指定的行动；否则会跳过行动。利用 *if/else*（假如 / 否则）选择结构，程序员可针对条件成立与不成立这两种情况，分别指定一项行动。例如下面的伪代码语句：

```
假如学生成绩大于或等于 60
    打印 "Passed"
否则
    打印 "Failed"
```

上例中，假如学生成绩大于或等于 60 分，则打印 "Passed"；如小于 60 分，则打印 "Failed"。不管哪种情况，都会在完成打印操作后，顺序执行下一条伪代码语句。注意在伪代码中，*else*（否则）的主体代码也进行了缩进。控制结构缩进的主体称为一个 *suite*。记住，在程序中应始终贯彻选择的缩进约定。缩进不正确，Python 就无法正确执行代码；另外，如程序不遵守统一的间距约定，也会导致代码难以阅读。

良好编程习惯 3.1 如果同时有几级缩进，每个 *suite* 都必须缩进。相同级别的不同 *suite* 不必具有相同的缩进量，但这是一种良好的编程习惯。

上述伪代码 *if/else* 结构可用 Python 写成：

```
if grade >= 60:
    print "Passed"
else:
    print "Failed"
```

常见编程错误 3.2 没有对从属于 *if* 或 *else* 这两种 suite 的全部语句进行正确缩进，会造成语法错误。

图 3.4 的流程图演示了 *if/else* 结构中的控制流程。再次重申，除小圆圈和箭头（流线）之外，流程图里还使用了矩形（标明行动）和菱形（标明决策）。后文将继续强调这种行动 / 决策计算模型。同样地，可想象一个柜子里包含了空的双选结构。程序员的任务是采用堆叠和嵌套方式，将这些选择结构同算法所需的其他控制结构合并到一起，再使用适合算法的行动与决策来填充矩形和菱形。

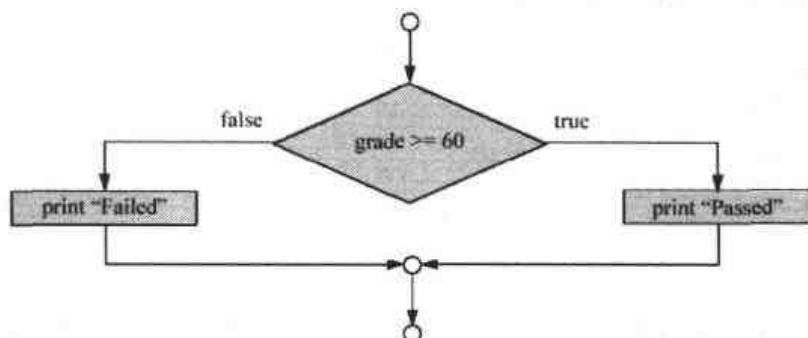


图 3.4 *if/else* 双选结构流程图

嵌套 *if/else* 结构检测多个条件，它的做法是将一个 *if/else* 选择结构放入另一个 *if/else* 选择结构。例如，下列伪代码语句为大于或等于 90 分的成绩打印 A；为 80~90 分之间的成绩打印 B；为 70~79 分之间的成绩打印 C；为 60~69 分之间的成绩打印 D；为其他成绩打印 F。

假如学生成绩大于或等于 90

打印“A”

否则

假如学生成绩大于或等于 80

打印“B”

否则

假如学生成绩大于或等于 70

打印“C”

否则

假如学生成绩大于或等于 60

打印“D”

否则

打印“F”

上述伪代码用 Python 写成，便是：

```
if grade >= 90:
    print "A"
else:
    if grade >= 80:
        print "B"
    else:
        if grade >= 70:
            print "C"
```

```

else:
    if grade >= 60:
        print "D"
    else:
        print "F"

```

如 grade 大于或等于 90，前 4 个条件都为 true，但只有第一次检测后的 print 语句才会执行。之后，外层 if/else 语句的 else 部分会被跳过。

性能提示 3.1 嵌套 if/else 结构比一系列单选 if 结构快，因为只要有一个条件满足，其余测试就会终止。

性能提示 3.2 在嵌套 if/else 结构中，把最可能成立的条件放在该嵌套 if/else 结构的开始处。和把不常见的条件放在开始处相比，采用这种做法后，嵌套 if/else 结构运行得更快。

许多 Python 程序员都喜欢将上述 if 结构写成：

```

if grade >= 90:
    print "A"
elif grade >= 80:
    print "B"
elif grade >= 70:
    print "C"
elif grade >= 60:
    print "D"
else:
    print "F"

```

这样可将双选 if/else 结构替换成多选 if/elif/else 结构。两种形式效果一样。后者之所以流行，是因为其避免了代码过分朝右边缩进。这样的缩进会在一行上留下较少的空间，造成不得不将一行分解成多行，从而降低了程序的可读性。

每个 elif 都可包含一项或多项行动。图 3.5 的流程图显示常规的 if/elif/else 多选结构。该流程图表明，执行了 if 或 elif 语句后，控制权会立即退出 if/elif/else 结构。再次重申，除了小圆圈和箭头之外，流程图里包含了矩形和菱形符号。想象程序员可访问一个大柜子，其中包含大量空的 if/elif/else 结构——具体数量由程序员决定，目的是与其他控制结构堆叠和嵌套，以便结构化地实现一个算法的控制流程。然后，用适合算法的行动和决策来填充矩形和菱形。

在 if/elif/else 结构中，else 语句是可选的。然而，大多数程序员都在一系列 elif 语句结束时插入一个 else 语句，以便处理同 elif 语句中指定的任何条件都不匹配的条件。我们将 else 语句所处理的条件称为“默认条件”。如 if/elif 结构指定了一个 else 语句，它就必须是结构中的最后一个语句。

良好编程习惯 3.2 要在 if/elif 结构中提供一个默认条件。无默认条件的 if/elif 结构中，没有被显式检测的条件会被忽略。包括一个默认条件，可强迫程序员处理异常的条件。

软件工程知识 3.3 在程序中，可以放入一个语句的任何地方都可放入一个 suite。

if 选择结构主体 (suite) 可包含几个语句，所有这些语句都必须缩进。下例中，if/else 结构的 else 部分包括了一个 suite，其中含有两个语句。包含多个语句的 suite 有时也称为“复合语句”。

```

if grade >= 60:
    print "Passed."
else:
    print "Failed."
    print "You must take this course again."

```

在这个例子中，如 grade 小于 60，程序会执行 else 中的两个语句，并打印：

```

Failed.
You must take this course again.

```

注意，else 这个 suite 的两个语句都进行了缩进。如果以下语句没有缩进：

```
print "You must take this course again."
```

无论成绩是否小于 60，它都会执行。这是“逻辑错误”的一个典型例子。

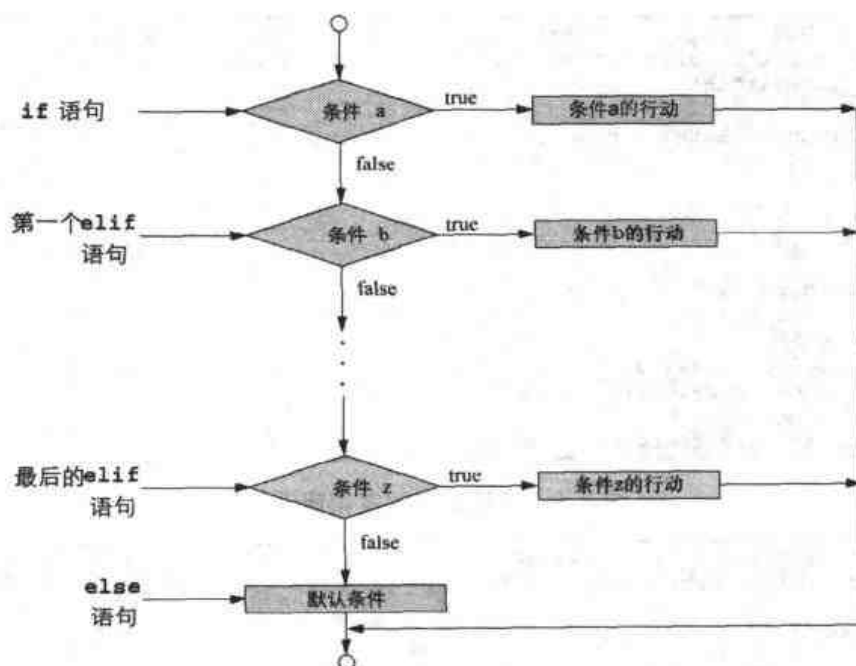


图 3.5 if/elif/else 多选结构

程序中可能出现两种主要错误：语法错误和逻辑错误。语法错误是指违反了程序语言的规则，例如将关键字用作标识符，或忘记在 if 语句后添加冒号 (:)。解释器能捕捉语法错误，并显示错误提示消息。

逻辑错误导致程序产生非预期的结果，而且不可由解释器捕捉。“严重逻辑错误”会导致程序失败，并提前终止。对于严重错误，Python 会打印一条错误消息（名为“traceback”），然后退出。非严重逻辑错误则允许程序继续执行，但会产生不正确的结果。

常见编程错误 3.3 忘记对 suite 中的所有语句进行缩进，会导致语法或逻辑错误。

图 3.6 的交互式会话试图对两个用户输入的值执行除法运算，并演示一个语法错误和两个逻辑错误。语法错误出现在下面这一行：

```
print value1 +
```

运算符+需要右操作数，所以解释器会报告语法错误。第一个逻辑错误出现在下面这一行：

```
print value1 + value2
```

目的是打印两个用户输入的整数值之和。但是，字符串没有先转换成整数，所以语句不会产生希望的结果。相反，语句会连接两个字符串——把两个字符串合到一起。注意，解释器没有显示任何消息，因为语句本身是合法的。

第二个逻辑错误出现在下面这一行：

```
print int( value1 ) / int( value2 )
```

由于未检查用户输入的第二个值是否为 0，所以程序可能出现除以零的情况，这属于严重逻辑错误。

常见编程错误 3.4 除以零是严重的逻辑错误。

以前说过，任何可放入单个语句的地方都可放入多个语句。除此之外，还有另一种可能，即什么语句都不放入（或者说“放入空语句”）。要表示空语句，请在正常语句出现的位置放入一个关键字 `pass`，如图 3.7 所示。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> value1 = raw_input("Enter a number: ")
Enter a number: 3
>>> value2 = raw_input("Enter a number: ")
Enter a number: 0
>>> print value1 +
      File "<stdin>", line 1
        print value1 +
              ^
SyntaxError: invalid syntax
>>> print value1 + value2
30
>>> print int(value1) / int(value2)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

图 3.6 语法错误和逻辑错误

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> if 1 < 2:
...     pass
...
```

图 3.7 关键字 `pass`

常见编程错误 3.5 所有控制结构至少要包含一个语句。不包含语句的控制结构会导致语法错误。

3.7 while 重复结构

“重复结构”可指定一项行动。只要某些条件保持 `true`，行动就会不断执行。例如以下伪代码语句：

只要（While）我的购物清单上还有商品
 购买下一种商品，并把它从清单上划掉

它描述了购物时的重复操作。其中，“我的购物清单上还有商品”是一个条件，既可能成立（`true`），也可能不成立（`false`）。如果为 `true`，就采取“购买下一种商品，并把它从清单上划掉”行动。只要条件为 `true`，就重复这一行动。

`while` 重复结构包含的语句构成了 `while` 的主体（`suite`）。`while` 结构主体可由一个或多个语句构成。最终，条件应变为 `false`（买了最后一件商品，并把它从清单上划掉）。在这个时候，重复会终止，并执行重复结构之后的第一个语句。

常见编程错误 3.6 在 `while` 结构主体中，如果没有提供最终会造成 `while` 条件变成 `false` 的一项行动，会造成“无限循环”这种逻辑错误——重复结构永远不能终止。

常见编程错误 3.7 如将关键字 `while` 拼成 `While`，会导致语法错误（记住 Python 是要区分大小写的）。所有 Python 保留关键字，比如 `while`、`if`、`elif` 和 `else` 等等，都只能采用小写字母。

下面来考虑 `while` 结构的一个例子，以下程序段用于计算 2 的所有乘方值，并判断哪个值首先大于 1000。在此，假定变量 `product` 创建并初始化成 2。下述 `while` 重复结构执行完毕之后，`product` 就会包含我们需要的答案：

```

product = 2
while product <= 1000:
    product = 2 * product

```

进入 while 结构时, product 为 2。product 变量不断地被 2 乘, 被连续指派值 4, 8, 16, 32, 64, 128, 256, 512 以及 1024。一旦 product 变成 1024, while 结构的条件 (product <= 1000) 就成为 false, 从而终止重复——product 的终值为 1024。程序将继续执行 while 之后的下一条语句。

图 3.8 的流程图揭示了与前述 while 结构对应的控制流程。同样地, 注意在流程图中, 除了小圆圈和箭头之外, 还包含一个矩形和一个菱形符号。

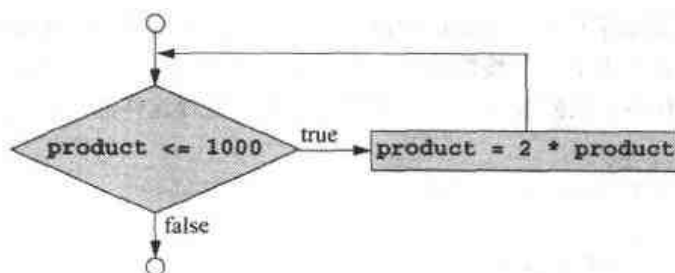


图 3.8 while 重复结构流程图

设想在一个柜子中, 包含了一系列空的 while 结构, 它们可堆叠在其他控制结构上面, 或与它们嵌套, 以实现算法的控制流程。随后, 可用适当的行动和决策来填充空矩形和菱形。该流程图清晰展示了重复过程。从矩形出发的流线会返回决策 (菱形), 除非决策结果为 false, 否则每次都要对决策进行检测。变成 false 后, 会立即退出 while 结构, 并将控制权移交给程序中的下一个语句。

3.8 算法陈述: 案例分析 1 (由计数器控制的重复)

为演示算法的开发, 下面来解决一个求平均成绩问题的几种变化形式。问题陈述如下:

全班 10 名学生参加一个测验。成绩 (0 到 100 之间的一个整数) 已经出来。请计算平均成绩。

平均成绩等于成绩总和除以学生人数。解决这个问题的算法是获得每个成绩, 执行求平均值计算, 再打印结果。

使用伪代码, 我们列出要采取的行动, 并指出它们的顺序。这里使用“由计数器控制的重复”来每次输入一个成绩。该技术要求使用一个名为 counter (计数器) 的变量, 用它控制一系列语句重复执行的次数。一旦计数器超过 10, 便终止重复。在本节, 我们展示了伪代码算法 (图 3.9), 以及相应的程序 (图 3.10)。在下一节, 还要展示如何开发伪代码算法。由计数器控制的重复通常称为“确定重复” (Definite Repetition)——因为在循环执行之前, 已经知道了要重复的次数。

```

将总和 (total) 设成 0
将成绩计数器 (counter) 设成 1

只要成绩计数器小于或等于 10
    输入下一个成绩
    将成绩加到总和中
    成绩计数器加 1

将平均成绩设为总和除以 10
打印平均成绩

```

图 3.9 使用“由计数器控制的重复”解决平均成绩问题的伪代码算法

伪代码算法中提到了一个总成绩 (total) 和一个计数器 (counter)。在图 3.10 的程序中, 变量 total

(第 5 行)用于累加一系列值;变量 `counter` 则负责计数——在这个例子中,它统计输入了多少个成绩。`total` 变量通常应初始化成零。

第 5~6 行是赋值语句,将 `total` 初始化为 0,将 `gradeCounter` 初始化为 1。第 9 行指出只要 `gradeCounter` 值小于或等于 10, `while` 结构就应继续。第 10~11 行对应伪代码语句“输入下一个成绩”。函数 `raw_input` 在屏幕上提示:“Enter grade:”,并接受用户输入。第 11 行将用户输入的字符串转换成整数。接着,程序使用新输入的 `grade` 更新 `total`——第 12 行将 `grade` 加到 `total` 的上一个值上,并将结果指派给 `total`。

然后,程序使变量 `gradeCounter` 递增,指出已处理了一个成绩。第 13 行使 `gradeCounter` 自增 1,使 `while` 结构的条件最终能变成 `false` 并终止循环。`while` 终止后,会执行第 16 行,并将平均值计算结果指派给变量 `average`。第 17 行显示字符串“Class average is”,后续一个空格(由 `print` 插入),最后是变量 `average` 的值。注意,求平均值计算产生了一个整数结果。实际上,这个例子的成绩总和为 817。它被 10 除后,结果应是 81.7,这是一个有小数点的数字。下一节将讨论如何处理这样的浮点数。

```
1 # Fig. 3.10: fig03_10.py
2 # Class average program with counter-controlled repetition.
3
4 # initialization phase
5 total = 0          # sum of grades
6 gradeCounter = 1   # number of grades entered
7
8 # processing phase
9 while gradeCounter <= 10:          # loop 10 times
10     grade = raw_input( "Enter grade: " ) # get one grade
11     grade = int( grade ) # convert string to an integer
12     total = total + grade
13     gradeCounter = gradeCounter + 1
14
15 # termination phase
16 average = total / 10          # integer division
17 print "Class average is", average
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

图 3.10 使用“由计数器控制的重复”解决平均成绩问题

良好编程习惯 3.3 初始化 `counter` 和 `total` 变量。

在图 3.10 的程序中,假如第 16 行使用 `gradeCounter` 而不是用 10 来执行计算,程序会输出不正确的 74,这是由于 `while` 循环终止后, `gradeCounter` 包含的值是 11。图 3.11 通过一个交互式会话演示了 `while` 循环重复 10 次后, `gradeCounter` 的值是多少。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> gradeCounter = 1
>>> while gradeCounter <= 10:
...     gradeCounter = gradeCounter + 1
...
>>> print gradeCounter
11
```

图 3.11 由计数器控制的循环终止后使用的计数器值

3.9 算法陈述，自上而下求精法：案例分析 2（由哨兵值控制的重复）

下面要对求平均成绩问题进行“泛化”。请考虑以下问题：

开发一个求平均成绩程序。该程序每次执行时，都能处理任意数量的成绩。

在第一个求平均成绩例子中，成绩的数量（10）事先已经知道。在下例中，则事先不假定要输入多少个成绩。程序必须能处理任意数量的成绩。那么，程序何时应该停止成绩输入呢？什么时候应该计算，什么时候则应打印平均成绩呢？

为解决这个问题，一个办法是使用名为“哨兵”的一个特殊值来指出“输入结束”。也可把这种值称为“信号值”、“假值”或“旗帜值”等。用户要不断输入成绩，直到所有成绩都输入完毕。然后，可输入一个特殊的哨兵值，指明已经输入了最后一个成绩。由哨兵值控制的重复通常称为“不确定重复”（Indefinite Repetition），因为在循环开始之前，并不知道重复次数。

显然，哨兵值的选择是有讲究的，否则可能与正式输入产生混淆。由于测验成绩通常都是非负的整数，所以-1对该问题来说是能接受的一个哨兵值。因此，执行一遍这个求平均成绩程序，可能得到像“95, 96, 75, 74, 89 和-1”这样的一连串输入。遇到-1，程序应能马上计算并打印出 95, 96, 75, 74 和 89 这 5 个值的平均值。

常见编程错误 3.8 选择一个合法数据作为哨兵值，会造成逻辑错误。

为生成求平均成绩程序，我们采用一种名为“自上而下求精法”的技术，它有助于开发具有良好结构的程序。首先用伪代码来表示“上部”：

判断测验的平均成绩

在上部，是一条简单语句，它概括了程序的总体功用。因此，最上部的语句实际是对程序的一个完整描述。但令人遗憾的是，顶部（如上所示）无法传达足够细节，无法据此写一个完整 Python 程序。所以，接下来要进行“求精”。首先，将上部的语句分解成一系列更小的任务，然后按它们的执行顺序排列好，这样便获得了“第一次求精”结果：

初始化变量

输入、总计和计数测验成绩

计算并打印平均成绩

这里只使用了顺序结构——上述步骤会顺序执行。

软件工程知识 3.4 每次求精都获得了算法的一个完整规范说明，只是细化程度有所区别。

软件工程知识 3.5 许多程序都可从逻辑上分解成 3 个阶段：初始化阶段（对变量进行初始化）；处理阶段（输入数据值，并相应地调节程序变量）和结束阶段（计算并打印最终结果）。

如果是第一次求精，参照上述“软件工程知识”进行操作便足够了。要进行下一级求精（第二次求精），需要用到一些特定的变量。程序要维持一个不断变化的 total 值，统计处理了多少个成绩的 count 值，包含了每个成绩值的一个变量，以及包含了计算好的平均成绩的一个变量。以下伪代码语句：

初始化变量

可“求精”为：

将总和初始化成零

将计数器初始化成零

以下伪代码语句：

输入、总计和计数测验成绩

则需要用一个重复（即“循环”）结构来实现，它能连续输入每个成绩。由于事先不知道要输入的成绩个数，所以计划用一个“由哨兵值控制的重复”。用户可不断输入合法的成绩值。最后一个合法成绩输入完毕后，在下次重复时，就输入哨兵值。程序会在每次输入成绩后检测是否为哨兵值，如答案是肯定的，就终止循环。上述伪代码语句的二次求精结果如下：

```

输入第一个成绩（可能是哨兵值）
只要（While）用户还没有输入哨兵值
    把这个成绩加到总和里
    成绩计数器自增 1
    输入下一个成绩（可能是哨兵值）

```

以下伪代码语句：

计算并打印平均成绩

可求精为：

```

假如（If）计数器不等于 0
    将平均值设为总和除以计数器值
    打印平均值
否则（else）
    打印“没有输入成绩”

```

注意这儿检测了“除以零”的可能。“除以零”属于严重逻辑错误。如果未能检测到它，程序就会失败（崩溃）。图 3.12 展示了求平均成绩问题伪代码的完整二次求精结果：

```

将总和初始化为零
将计数器初始化为零

输入第一个成绩（可能是哨兵值）
只要（While）用户还没有输入哨兵值
    把这个成绩加到总和里
    成绩计数器自增 1
    输入下一个成绩（可能是哨兵值）

假如（If）计数器不等于 0
    将平均值设为总和除以计数器值
    打印平均值
否则（else）
    打印“没有输入成绩”

```

图 3.12 用哨兵值控制的重复来解决平均成绩问题的伪代码算法

良好编程习惯 3.4 执行除法运算时，如除数可能为零，请务必明确检测，并在程序中进行相应的处理（比如打印一条错误信息）。不要任由严重错误发生。在第 12 章，我们还会具体讨论如何使程序识别此类错误，并采取相应的行动。这称为“异常处理”。

图 3.9 和图 3.12 的伪代码中包含一些空行，目的是改善伪代码的可读性。这些空行清楚地将这些语句分为不同的执行阶段。

图 3.12 的伪代码算法用于解决更“泛化”的求平均成绩问题。算法是在经两次“求精”后开发出来的。有些情况下，可能还要进一步求精。

软件工程知识 3.6 只要伪代码算法提供了足够细节,利用这些细节可将伪代码轻松转换成 Python 程序,便应停止“自上而下求精”。之后,即可轻松根据伪代码写一个 Python 程序。

图 3.13 展示了这个 Python 程序及其示范执行。尽管输入的是整数成绩,但平均值计算可能生成小数(实数)。整数数据类型不能表示实数,所以程序用浮点数据类型来处理小数,并引入了 float 函数,它强迫求平均值计算生成一个浮点数结果。

```

1 # Fig. 3.13: fig03_13.py
2 # Class average program with sentinel-controlled repetition.
3
4 # initialization phase
5 total = 0          # sum of grades
6 gradeCounter = 0   # number of grades entered
7
8 # processing phase
9 grade = raw_input( "Enter grade, -1 to end: " ) # get one grade
10 grade = int( grade ) # convert string to an integer
11
12 while grade != -1:
13     total = total + grade
14     gradeCounter = gradeCounter + 1
15     grade = raw_input( "Enter grade, -1 to end: " )
16     grade = int( grade )
17
18 # termination phase
19 if gradeCounter != 0:
20     average = float( total ) / gradeCounter
21     print "Class average is", average
22 else:
23     print "No grades were entered"

```

```

Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.5

```

图 3.13 用于解决平均成绩问题的、由哨兵值控制的重复

本例证明,控制结构能一个接一个地堆叠起来(按顺序),就像小孩子搭积木那样。while 结构(第 12~16 行)后面紧跟一个 if/else 结构(第 19~23 行)。程序的许多代码与图 3.10 完全一致,所以这里只讲解其中的新特性和新问题。

第 6 行将变量 gradeCounter 初始化成 0,这是由于当前未输入任何成绩。为准确记录输入成绩数量,只有在输入一个成绩值后,变量 gradeCounter 才应增值。

良好编程习惯 3.5 在由哨兵值控制的循环中,当提示输入时,应明确指出哨兵值是哪一个。

现在研究一下由哨兵值控制的重复(图 3.13)和由计数器控制的重复(图 3.10)对程序逻辑造成的差异。在由计数器控制的重复中,每遍历一次 while 结构(遍历次数是事先指定的),都要从用户处读取一个值。在由哨兵值控制的重复中,是在程序抵达 while 结构之前读取一个值(第 9~10 行)。这个值决定了程序控制流程是否应该进入 while 结构主体。假如 while 结构的条件为 false(表明用户已输入哨兵值),程序不再执行 while 循环(因为没有输入实际的成绩)。与此相反,假如条件为 true,程序会执行 while 循环,并处理用户输入的值(即将 grade 加到 total 上)。处理完这个 grade 后,程序请求用户输入下一个 grade。执行 while 循环的最后一行(缩进的那一行)后,即第 16 行,会使用刚才用户输入的新值,再次检测 while 结构条件,判断是否应该再次执行 while 主体。注意,程序是在对 while 结构进行求

值之前请求下一个值。这样一来，就能先判断刚才输入的值是否为哨兵值，再对该值进行处理（即加到 total 上）。如果输入的是哨兵值，while 结构会终止，新输入的值不会加到 total 上。

第 9~10 行和第 15~16 行包含完全相同的代码。在 3.15 节，我们将介绍能避免重复代码的技术。

平均值并非一定是整数值。平均值一般都包含小数部分，比如 7.2 或 93.5。这些值称为“浮点数”。

total / gradeCounter 计算会得到一个整数结果，因为 total 和 counter 都包含整数值。两个整数相除叫做整数除法，根据 Python 的约定，结果中的任何小数部分都会被删除（即进行“截尾”处理）。注意，小数部分是在结果指派给 average 之前被截去的。要通过整数值生成浮点值结果，需要用 float 函数将一个或两个值转换成浮点值。记住，函数本质上是执行特定任务的代码片断；在第 20 行，float 函数将变量 sum 的整数值转换成浮点值。现在计算的是一个被整数 gradeCounter 除的浮点值。

在表达式中，如果操作数的数据类型完全一致，Python 解释器知道怎样对其进行求值。为了保证操作数具有相同的类型，解释器会对选择的操作数执行“提升”操作，这也称为“隐式转换”。例如，在同时包含整数和浮点数据的表达式中，整数操作数会被“提升”为浮点数。在我们的例子中，gradeCounter 的值会提升成一个浮点数。之后才会执行计算，并将浮点除法运算的结果指派给变量 average。

常见编程错误 3.9 将所有浮点数都假设为是精确的，会导致不正确的结果。浮点数在大多数计算机中只是近似值。

尽管浮点数并不精确，但的确很有用。例如，当我们说人体正常体温是 98.6 华氏度时，不需要精确到许多位。用一支温度计查看温度时，虽然读数是 98.6，但实际可能是 98.5999473210643。总之，数字虽然只是 98.6，但足以满足我们的需要。

浮点数只是“近似值”可通过除法运算加以证明。10 除以 3，结果是 3.3333333...。一系列 3 会无限重复下去。然而，计算机只分配了一个固定大小的空间来容纳这样的值，所以很明显，浮点数只是近似值。

3.10 算法陈述，自上而下求精法：案例分析 3（嵌套控制结构）

再来研究另一个完整的例子。同样，先用伪代码技术和“自上而下求精法”设计一个算法，再根据它写出相应的 Python 程序。下面是问题陈述：

一所大学提供一门课程，让学生准备参加本州房地产经纪人资格考试。去年完成这门课的几名学生参加了资格考试。自然，这所大学想知道自己的学生在考试时的表现。现在，要求您写一个程序，对考试结果进行总结。您得到 10 名学生的一个列表。在每个姓名旁边，1 表示该学生通过了考试；2 表示没有通过。

程序应该像下面这样分析考试结果：

1. 输入每一个考试结果（1 或 2）。每次请求另一个考试结果时，都在屏幕上显示消息：“Enter result”（输入结果）。
2. 统计两类考试结果的数量（1 的数量和 2 的数量）。
3. 显示考试结果总结，分别指出通过和没有通过考试的学生的人数。
4. 假如有 8 名以上的学生通过考试，便打印一条消息“Raise tuition”（提高学费）。

仔细阅读上述问题陈述，可得出以下几个结论：

1. 程序必须处理 10 个考试结果，所以应该使用一个由计数器控制的循环。
2. 每个考试结果都是一个数字，要么是 1，要么是 2。程序每次读入一个考试结果，都必须检测该数字是 1 还是 2。在我们的算法中，打算检测的是 1。假如数字不是 1，则假定它是 2。
3. 要使用两个计数器：一个统计通过考试的学生数量，另一个统计没有通过的数量。

4. 程序处理完所有结果后, 必须判断是否有 8 名以上的学生通过了考试。

现在开始运用“自上而下求精法”, 首先是上部的伪代码表示:

分析考试结果, 判断是否该提高学费

同样地, 它虽然是对程序的完整表示, 但还要进一步求精, 否则无法将伪代码自然地转换成 Python 程序。第一次求精结果是:

初始化变量

连续输入 10 个考试成绩, 统计通过和没有通过考试的学生人数

打印考试结果总结, 判断是否该提高学费

显然, 以上的信息还不够“细”。虽然获得了程序的完整表示, 但仍需进一步求精。先考虑一系列特定的变量。我们要用两个计数器记录通过和未通过的学生人数, 要用一个计数器控制循环, 还要用一个变量保存用户输入。以下伪代码语句:

初始化变量

可求精为:

把 passes (通过考试) 初始化成 0

把 failures (未通过考试) 初始化成 0

把 studentCounter (学生计数器) 初始化成 1

注意, 上面只初始化了用于统计通过、未通过和学生人数的计数器。以下伪代码语句:

连续输入 10 个考试成绩, 统计通过和没有通过考试的学生人数

需要用一个循环来连续输入每个考试成绩。由于提前知道需要 10 个考试成绩, 所以当然应该使用一个由计数器控制的循环。在循环内部 (它嵌套于另一个循环内), 用一个双选结构判断每个考试结果到底是通过, 还是没有通过。然后, 让相应的计数器自增 1。对上述伪代码语句的二次求精结果是:

假如 (While) 学生计数器小于或等于 10

输入下一个考试结果

假如 (If) 学生通过考试

在 passes 上加 1

否则 (Else)

在 failures 上加 1

在学生计数器上加 1

注意用空行将 If/else 控制结构独立出来, 以改善程序可读性。以下伪代码语句:

打印考试结果总结, 判断是否该提高学费

可求精为:

打印通过考试的人数 (passes)

打印没有通过考试的人数 (failures)

假如 8 名以上的学生通过

打印 "Raise tuition" (提高学费)

图 3.14 展示了完整的二次求精结果。注意, 伪代码也用空行将 while 结构独立出来, 以改善程序的可读性。

把 passes (通过考试) 初始化成 0
把 failures (未通过考试) 初始化成 0
把 studentCounter (学生计数器) 初始化成 1

假如 (While) 学生计数器小于或等于 10
 输入下一个考试结果

 假如 (If) 学生通过考试
 在 passes 上加 1
 否则 (Else)
 在 failures 上加 1
 在学生计数器上加 1

打印通过考试的人数 (passes)
打印没有通过考试的人数 (failures)

假如 8 名以上的学生通过
 打印 "Raise tuition" (提高学费)

图 3.14 考试结果问题的伪代码

伪代码已充分进行了求精, 可方便地转换成 Python。图 3.15 展示了 Python 程序以及两次示范执行结果。

注意, 第 4 行使用相等运算符 (==) 检测变量 result 的值是否等于 1。千万不要混淆相等运算符和赋值运算符 (=), 否则容易造成语法或逻辑错误。

常见编程错误 3.10 在条件语句中用符号=来判断相等性是语法错误。

常见编程错误 3.11 用运算符==赋值是逻辑错误。

软件工程知识 3.7 经验表明, 用计算机解决问题最有效的办法是为解决方案开发一种算法。一旦开发出正确的算法, 通常能根据它方便地生成一个能实际工作的 Python 程序。

软件工程知识 3.8 许多有经验的程序员在写程序之前, 从来不用伪代码这样的程序开发工具。他们觉得自己的最终目标是用计算机解决问题, 写伪代码会推迟进度。尽管对于简单和熟悉的程序可以这样做, 但在从事大型的、复杂的项目时, 这样做有可能导致严重错误, 反而会推迟进度。

```
1 # Fig. 3.15: fig03_15.py
2 # Analysis of examination results.
3
4 # initialize variables
5 passes = 0           # number of passes
6 failures = 0         # number of failures
7 studentCounter = 1   # student counter
8
9 # process 10 students; counter-controlled loop
10 while studentCounter <= 10:
11     result = raw_input("Enter result (1=pass,2=fail): ")
12     result = int(result) # one exam result
13
14     if result == 1:
15         passes = passes + 1
16     else:
17         failures = failures + 1
18
19     studentCounter = studentCounter + 1
20
```

```

21 # termination phase
22 print "Passed", passes
23 print "Failed", failures
24
25 if passes > 8:
26     print "Raise tuition"

```

```

Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition

```

```

Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Passed 6
Failed 4

```

图 3.15 考试结果问题

3.11 增量赋值符号

Python 提供几个增量赋值符号，用于简写赋值表达式。例如以下语句：

```
c = c + 3
```

使用“增量加法赋值符号”`+=`，可把它简写成：

```
c += 3
```

`+=`符号将符号右边的表达式的值加到左边的变量值上，再将结果存回左边的变量。任何具有以下形式的语句（其中的“运算符”是指一个二元运算符，比如`+`，`-`，`**`，`*/`或`%`）

变量 = 变量 运算符 表达式

都可简写成

变量 运算符 = 表达式

使用了增量赋值符号的语句称为“增量赋值语句”。图 3.16 总结了增量算术赋值符号。

移植性提示 3.1 Python 2.0 版本开始引入增量赋值符号，在老版本 Python 中使用增量赋值符号是语法错误。

常见编程错误 3.12 在赋值符号左边的变量初始化之前试图使用增量赋值是错误的。

运算符	示范表达式	解释	赋值
假定: c=3, d=5, e=4, f=2, g=6, h=12			
+=	c += 7	c = c + 7	10 指派给 c
-=	d -= 4	d = d - 4	1 指派给 d
*=	e *= 5	e = e * 5	20 指派给 e
**=	f **= 3	f = f ** 3	8 指派给 f
/=	g /= 3	g = g / 3	2 指派给 g
%=	h %= 9	h = h % 9	3 指派给 h

图 3.16 增量算术赋值符号

3.12 由计数器控制的重复的本质

由计数器控制的重复需要:

1. 一个控制变量的名称 (或循环计数器)
2. 控制变量的初始值
3. 每次循环时, 控制变量的自增或自减量
4. 用于检测控制变量终值的条件 (也就是循环是否应该继续)

图 3.17 的程序打印从 0~9 的数字。第 4 行命名控制变量 (counter), 并把初始值设为 0。在第 8 行, while 结构针对每次循环都使 counter 自增 1。while 结构的循环继续条件检测 counter 的值是否小于 10。如果大于或等于 10, 循环终止。

```

1 # Fig. 3.17: fig03_17.py
2 # Counter-controlled repetition.
3
4 counter = 0
5
6 while counter < 10:
7     print counter
8     counter += 1

```

```

0
1
2
3
4
5
6
7
8
9

```

图 3.17 由计数器控制的重复

常见编程错误 3.13 由于浮点值可能是近似值, 所以如果用浮点变量控制循环计数, 可能导致不准确的计数器值, 不能准确检测终止条件。程序应该用整数值控制循环计数。

良好编程习惯 3.6 在每个控制结构前后各留一个空行, 将其同程序的其余部分区分开。

良好编程习惯 3.7 嵌套级别过多, 会使程序难以理解。通常应将嵌套控制在 3 级以内。

良好编程习惯 3.8 在每个控制结构上下留一个空行，并对每个控制结构的主体进行缩进，使程序具有清晰的二维外观，增强可读性。

3.13 for 重复结构

for 重复结构处理由计数器控制的重复。为展示 for 的功能，下面改写图 3.17，结果见图 3.18。

```
1 # Fig. 3.18: fig03_18.py
2 # Counter-controlled repetition with the
3 # for structure and range function.
4
5 for counter in range( 10 ):
6     print counter
```

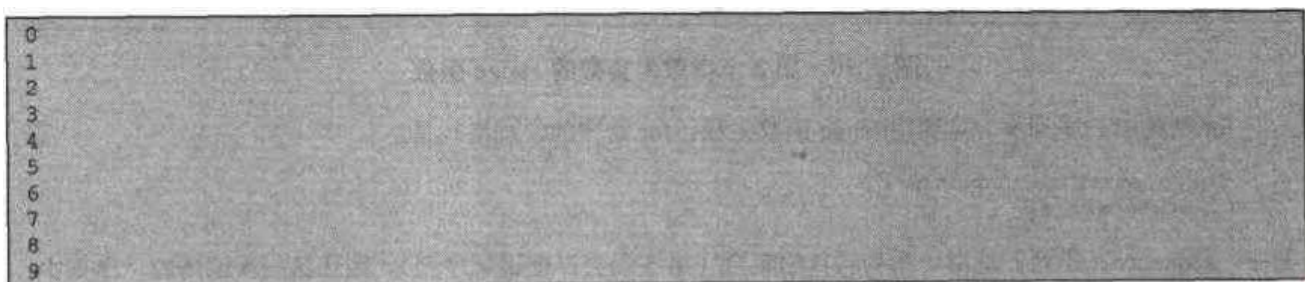


图 3.18 用 for 结构进行由计数器控制的重复

程序工作原理如下：for 结构开始执行，函数 range 会创建一个值的序列，范围为 0~9（图 3.19）。序列的第一个值指派给变量 counter，for 结构主体（第 6 行）开始执行。在序列中，后续每个值都会指派给变量 counter，并执行 for 结构主体。这个过程会一直重复，直至序列处理完毕。

图 3.19 显示了由函数 range 返回的序列。该序列是一个 Python 列表，其中包含 0~9 的整数。注意，列表中的值用方括号封闭（[]），并用逗号分隔。列表详情在第 5 章介绍。

注意，函数 range 返回的序列的最后一个值要比传给函数的参数值小 1。如错误地写成：

```
for counter in range( 9 ):
    print counter
```

循环执行 9 次。这是一种常见的逻辑错误，即“相差 1”错误。函数 range 可取得一个、二个或三个参数。向函数传递一个参数（图 3.19），那个参数（叫做 end）要比序列的上界（最高的值）大 1。在这种情况下，range 将返回以下范围的一个序列：

0 到 (end - 1)

如传递两个参数，第一个是 start，代表序列下界（返回序列中最低的值）；第二个参数是 end。在这种情况下，range 将返回以下范围的一个序列：

(start) 到 (end - 1)

如传递 3 个参数，前两个参数分别是 start 和 end，第三个参数是 increment，代表“自增值”。在这种情况下，调用 range 所生成的序列会从 start 到 end 递增，每次都递增或递减固定的值——如果 increment 值为正数，会递增；如果为负数，则会递减。以下 3 个 range 调用会生成相同的序列，如图 3.19 所示。图 3.20 则展示了一个递减序列的例子。

```
range( 10 )
range( 0, 10 )
range( 0, 10, 1 )
```

常见编程错误 3.14 如果忘记 range 函数返回的序列的第一个值是 0（前提是提供了下界），可能导致值

相差 1 的错误。

常见编程错误 3.15 如果忘记 range 函数返回的序列的最后一个值比函数的 end 参数值小 1，可能导致值相差 1 的错误。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

图 3.19 range 函数

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> range(10, 0, -1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

图 3.20 第 3 个参数为负数的 range 函数

for 结构中的序列不一定要用 range 函数生成。for 结构的常规格式是：

```
for element in sequence:
    statement(s)
```

其中，sequence（序列）是指一系列项目的集合（第 5 章会详细讲解序列）。循环第一次遍历时，序列中的第一项会指派给变量 element，并执行 statement（语句）。以后每次循环时，都先将序列中的下一项指派给变量 element，再执行 statement。为序列中的每一项都执行了一次循环后，循环会终止。大多数情况下，for 结构可用等价的 while 结构表示，例如：

```
initialization

while loopContinuationTest:
    statement(s)
    increment
```

其中，initialization（初始化）表达式对循环的控制变量进行初始化；loopContinuationTest 是循环继续条件，而 increment 使控制变量增值。

常见编程错误 3.16 创建一个没有主体语句的 for 结构是语法错误。

如果 for 结构的 sequence 部分是空白的（即序列里不包含值），程序不会执行 for 结构的主体，而是从 for 结构之后的语句继续执行。

程序有时会在循环主体中显示控制变量（element），或在计算时用到它。然而，并非一定要这样做。常见的做法是，仅用控制变量来控制重复，for 结构主体中则根本不用它。

良好编程习惯 3.9 避免在 for 循环主体更改控制变量的值，这有可能导致不易发现的逻辑错误。

for 结构的流程图和 while 差不多。图 3.21 展示了以下 for 语句的流程图：

```
for x in y:
    print x
```

流程图显示了初始化和更新过程。程序每次执行了主体语句后，都会进行更新。除小圆圈和箭头，流程图只包含矩形和菱形符号。程序员要在其中填充适合算法的行动和决策。

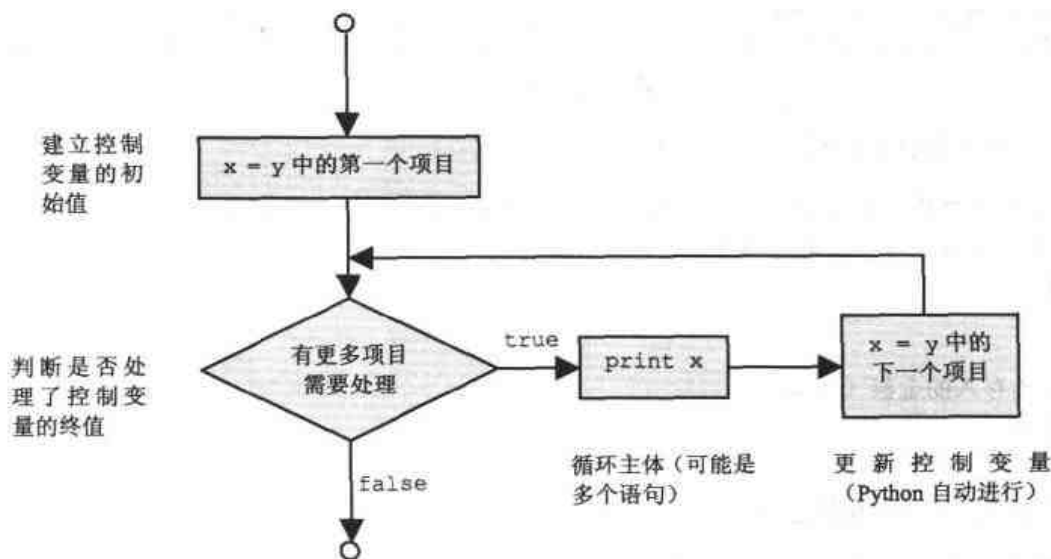


图 3.21 for 重复结构流程图

3.14 使用 for 重复结构

下例展示了在 for 结构中改变控制变量（循环计数器）的各种方法。每每遇到其中的各种情况，我们都会写下相应的 for 头部语句。注意，对 range 的第 3 个参数进行修改，可使控制变量递减。

a) 控制变量从 1 变成 100，每次增加 1。

```
for counter in range( 1, 101 ):
```

b) 控制变量从 100 变成 1，每次增加-1（减小 1）。

```
for counter in range( 100, 0, -1 ):
```

c) 控制变量从 7 变成 77，每次增加 7。

```
for counter in range( 7, 78, 7 ):
```

d) 控制变量从 20 变成 2，每次减小 2。

```
for counter in range( 20, 1, -2 ):
```

e) 控制变量为多个值的一个序列：2, 5, 8, 11, 14, 17, 20。

```
for counter in range( 2, 21, 3 ):
```

f) 控制变量为多个值的一个序列：98, 88, 77, 66, 55, 44, 33, 22, 11, 0。

```
for counter in range( 99, -1, -11 ):
```

下面两个例子演示 for 结构的简单应用。图 3.22 的程序用 for 结构求 2~100 的所有偶数整数的和。

```

1 # Fig. 3.22: fig03_22.py
2 # Summation with for.
3
4 sum = 0
5
6 for number in range( 2, 101, 2 ):
7     sum += number
8
9 print "Sum is", sum

```

Sum is 2550

图 3.22 用 for 求和

下例将用 for 结构计算复利。问题陈述如下：

某人新开一个户头，存入 1000 美元，年利率 5%。假定所有利息收入都重新存入户头，计算并打印在为期 10 年的时间里，每年结束时的存款金额。计算公式为：

$$a = p(1 + r)^n$$

其中：

p 是最开始存入的金额（本金）

r 是年利率

n 是存款年数

a 是在第 n 年结束时的存款金额

下面用循环来解决这个问题，它将计算在 10 年中每年累积的存款金额。图 3.23 展示了具体的程序。for 结构将循环主体执行 10 次，控制变量 year 从 1 递增至 10。在这个例子中，表达式 $(1 + r)^n$ 写成 $(1 + \text{rate})^{**} \text{year}$ 。其中，变量 rate 代表 r ，变量 year 代表 n 。

```
1 # Fig. 3.23: fig03_23.py
2 # Calculating compound interest.
3
4 principal = 1000.0 # starting principal
5 rate = .05         # interest rate
6
7 print "Year %21s" % "Amount on deposit"
8
9 for year in range( 1, 11 ):
10     amount = principal * ( 1.0 + rate ) ** year
11     print "%4d%21.2f" % ( year, amount )
```

Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.63
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

图 3.23 用于计算复利的 for 结构

for 循环之前的输出语句（第 7 行）和 for 循环中的输出语句（第 11 行）共同打印变量 year 和 amount 的值，并采用由 % 格式化运算符规范所指定的格式。字符 %4d 提出 year 列要用一个宽度为 4 的字段打印（也就是说，打印的值至少要占据 4 个字符位置）。如果要输出的值小于 4 个字符位置，就默认在字段中右对齐。如果要输出的值超过 4 个字符位置，就自动扩展字段宽度，以适应整个值的宽度。

字符 %21.2f 指出变量 amount 要打印成一个浮点值（由字符 f 指定），其中包含小数点。此列的总字段宽度是 21 个字符位置，而且小数点后有 2 位小数；在总字段宽度里，包括小数点和它右边的 2 个小数位；所以，小数点左边有 18 个位置。

注意，变量 amount、principal 和 rate 是浮点值。这样只是为了简化问题，因为要处理带有小数的美元金额，所以需要一种允许使用小数的类型。令人遗憾的是，这有时会造成问题。不妨举例说明用浮点值表示美元金额会造成什么问题（假定美元金额的小数点右侧显示 2 个数位）：假定在机器中保存的两个美元金额是 14.234（打印成 14.23）和 18.673（打印成 18.67）。两个金额相加，内部求和应得到 32.907，

打印时一般取近似值 32.91。所以打印结果可能是：

```
14.23
+18.67
-----
32.91
```

但是自己加一加，就知道结果应该是 32.90。务必注意这个问题！

良好编程习惯 3.10 用浮点值来执行财务方面的计算时，可一定要谨慎，近似值错误可能会导致不希望的结果。

注意，for 结构主体包含了 $1.0 + \text{rate}$ 这一计算（第 10 行）。实际上，每次循环时，该计算都会产生相同的结果，重复计算纯属浪费。更好的方案是定义一个变量（例如 `finalRate`），它在 for 结构开始前引用 $1.0 + \text{rate}$ 的值。然后，将第 10 行的 $1.0 + \text{rate}$ 替换成变量 `finalRate`。

性能提示 3.3 不要在循环内放入值不发生变化的表达式。

3.15 break 和 continue 语句

`break` 和 `continue` 语句用于更改控制流程。`break` 语句在 `while` 或 `for` 结构中执行时，会导致立即退出那个结构。程序继续执行结构之后的第一个语句。图 3.24 演示了在一个 `for` 重复结构中使用 `break` 语句的情况。if 结构一旦检测到 `x` 等于 5，就会执行 `break`。这样会终止 `for` 语句，并继续执行 `print` 语句（第 11 行）。循环将输出 4 个数字。

```
1 # Fig. 3.24: fig03_24.py
2 # Using the break statement in a for structure.
3
4 for x in range( 1, 11 ):
5
6     if x == 5:
7         break
8
9     print x,
10
11 print "\nBroke out of loop at x =", x
```

```
1 2 3 4
Broke out of loop at x = 5
```

图 3.24 for 结构中使用的 break 语句

图 3.25 是图 3.13 的一个修改版本，这个版本消除了老版本程序中的重复代码。第 9 行开始一个无限 `while` 循环。循环条件永远为 `true`，因为 1 始终代表 `true`。第 10~11 行提示用户输入成绩，并将输入转换成整数。如成绩是哨兵值 -1，就退出循环（第 14~15 行）。

```
1 # Fig. 3.25: fig03_25.py
2 # Using the break statement to avoid repeating code
3 # in the class-average program.
4
5 # initialization phase
6 total = 0          # sum of grades
7 gradeCounter = 0   # number of grades entered
8
9 while 1:
10     grade = raw_input( "Enter grade, -1 to end: " )
11     grade = int( grade )
12
13     # exit loop if user inputs -1
```

```

14     if grade == -1:
15         break
16
17     total += grade
18     gradeCounter += 1
19
20 # termination phase
21 if gradeCounter != 0:
22     average = float( total ) / gradeCounter
23     print "Class average is", average
24 else:
25     print "No grades were entered"

```

```

Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.5

```

图 3.25 用 break 语句消除重复代码

continue 语句在 while 或 for 结构中执行时，会跳过结构中其余的语句，继续下一次循环。在 while 结构中，循环继续检测会在 continue 语句执行之后立即进行。在 for 结构中，会将序列的下一个值指派给控制变量（前提是序列包含更多的值）。前面说过，通常可用 while 结构来表示 for 结构。但是，如果 while 结构中的自增表达式跟在 continue 语句之后，就不能这么做。在这种情况下，在检测循环继续条件之前，不会执行自增，所以 while 的执行方式有别于 for。图 3.26 在一个 for 结构中使用 continue 语句跳过结构中的输出语句，直接重复下一次循环。

```

1 # Fig. 3.26: fig03_26.py
2 # Using the continue statement in a for/in structure.
3
4 for x in range( 1, 11 ):
5
6     if x == 5:
7         continue
8
9     print x,
10
11 print "\nUsed continue to skip printing the value 5"

```

```

1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5

```

图 3.26 在 for 结构中使用 continue 语句

良好编程习惯 3.11 有的程序员认为 break 和 continue 违反了结构化编程准则。由于可采用后文即将讨论的结构化编程技术达到同样的目的，所以这些程序员不使用 break 和 continue。

3.16 逻辑运算符

到目前为止学习的都是“简单条件”，比如 counter <= 10, total > 1000 以及 number != sentinelValue 等。我们用关系运算符 >, <, >= 和 <= 以及相等运算符 == 和 != 表示这些条件。每项决策检测的都只是一个条件。如需检测多个条件才能做出一项决策，就要用单独的语句执行这些检测，或者通过嵌套 if 或 if/else 结构进行。Python 提供了“逻辑运算符”，可合并简单条件而构成复杂条件。逻辑运算符包括 and（逻辑 AND），or（逻辑 OR）以及 not（逻辑 NOT，也叫做逻辑非）。下面分别举例说明。

选择特定的执行路径前，假定想确保两个条件都为 true，那就可以像下面这样使用逻辑 and 运算符：

```
if gender == "Female" and age >= 65:
    seniorFemales += 1
```

if 语句实际包含两个简单条件。条件 `gender == "Female"` 判断一个人是否女性；条件 `age >= 65` 判断一个人是否为高龄居民。首先求值 and 运算符左边的简单条件，因为 `==` 的优先级高于 and。如有必要，接着会求值 and 运算符右边的简单条件，因为 `>=` 的优先级高于 and（后面会讲到，只有逻辑 AND 表达式左侧求值结果为 true，才会对右侧求值）。然后，if 语句会对以下复合条件进行判断：

```
gender == "Female" and age >= 65
```

只有两个简单条件都为 true，上述复合条件才会为 true。最后，如果该复合条件确实为 true，那么 `seniorFemales` 的计数会自增 1。假如两个简单条件有一个或全部为 false，程序会跳过自增运算，执行 if 之后的下一条语句。加上冗余括号之后，上述复合条件显得更易懂：

```
( gender == "Female" ) and ( age >= 65 )
```

图 3.27 中的表总结了 and 运算符。它针对表达式 1 和表达式 2，列出了 4 种可能的真假组合。这种表称为“真值表”（Truth Table）。

表达式 1	表达式 2	表达式 1 and 表达式 2
假	假	假
假	真	假
真	假	假
真	真	真

图 3.27 and（逻辑 AND）运算符真值表

任何表达式只要包含关系运算符和相等运算符，Python 都会将该表达式求值为真或假。为 false 的简单条件（例如 `age >= 65`）会求值为整数值 0；为 true 的简单条件会求值为整数值 1。求值为 0 的 Python 表达式为 false；求值为非 0 的整数值的 Python 表达式为 true，图 3.28 的交互式会话演示了这些概念。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more informa-
tion.
>>> if 0:
...     print "0 is true"
... else:
...     print "0 is false"
...
0 is false
>>> if 1:
...     print "non-zero is true"
...
non-zero is true
>>> if -1:
...     print "non-zero is true"
...
non-zero is true
>>> print 2 < 3
1
>>> print 0 and 1
0
>>> print 1 and 3
3
```

图 3.28 真值

交互式会话的第 5~10 行显示值 0 为 false。第 11~18 行显示任何非零的整数值都是 true。第 19 行

的简单条件求值为 `true` (第 20 行)。第 21 行和第 23 行的复合条件演示了 `and` 运算符的返回值。如果复合条件求值为 `false` (第 21 行), `and` 运算符就返回求值为 `false` 的第一个值 (第 22 行)。相反, 如复合条件求值为 `true` (第 23 行), `and` 运算符就返回条件中的最后一个值 (第 24 行)。

现在来看看 `or` (逻辑 OR) 运算符。在程序的某个位置采取特定行动之前, 如希望保证两个条件之一或者全部为 `true`, 就可使用 `or` 运算符。如下所示:

```
if semesterAverage >= 90 or finalExam >= 90:
    print "Student grade is A"
```

上述条件也合并了两个简单条件。简单条件 `semesterAverage >= 90` 判断学生是否可因为他在学期中一贯的良好表现而得“A”; 简单条件 `finalExam >= 90` 则判断是否由于学生的期末成绩大于或等于 90 分而得“A”。然后, `if` 语句考察复合条件:

```
semesterAverage >= 90 or finalExam >= 90
```

只要满足其中任何一个条件, 或两个条件全满足, 学生就可得“A”。注意, 假如上述两个简单条件都为 `false`, 就不打印“Student grade is A”消息。图 3.29 是 `or` 运算符的真值表。

表达式 1	表达式 2	表达式 1 or 表达式 2
假	假	假
假	真	真
真	假	真
真	真	真

图 3.29 `or` (逻辑 OR) 运算符真值表

如果复合条件求值为 `true`, `or` 运算符就返回求值为 `true` 的第一个值。相反, 如果复合条件求值为 `false`, `or` 运算符就返回条件中的最后一个值。

`and` 的优先级高于 `or`。两个运算符都从左到右顺序关联。包含 `and` 或 `or` 运算符的表达式会不断求值, 一直到能确定真假为止。这称为“短路求值”。因此, 对以下表达式的求值:

```
gender == "Female" and age >= 65
```

会在 `gender` 不等于“Female”的前提下立即停止, 因为已能确定整个表达式为 `false`。但假如 `gender` 等于“Female”, 就会继续求值 (因为如果条件 `age >= 65` 为 `true`, 整个表达式会为 `true`)。

性能提示 3.4 在使用了 `and` 运算符的表达式中, 假如不同条件是相互独立的, 就将最有可能为 `false` 的条件放在最左边。在使用了 `or` 运算符的表达式中, 要把最有可能为 `true` 的条件放在最左边。这样做可缩短程序执行时间。

Python 还提供了 `not` (逻辑非) 运算符, 用于反转一个条件的含义。`and` 和 `or` 运算符对两个条件进行合并 (二元运算符), 而逻辑非运算符只需要一个条件作为操作数 (也就是说, `not` 是一元运算符)。逻辑非运算符要放在希望反转的条件之前。如果希望在原始条件为 `false` 的前提下选择一个执行路径, 就可考虑使用逻辑非运算符, 如下例所示:

```
if not grade == sentinelValue:
    print "The next grade is", grade
```

图 3.30 是逻辑非运算符的真值表。许多时候, 程序员可避免使用逻辑非, 具体做法是采用合适的关系运算符或相等运算符, 以不同方式表示条件。例如, 上例可改写成:

```
if grade != sentinelValue:
    print "The next grade is", grade
```

借助于这种灵活性，程序员通常可采用更自然、更方便的方式来表示一个条件。

表达式	not 表达式
假	真
真	假

图 3.30 not（逻辑非）运算符真值表

图 3.31 总结了前面已介绍过的 Python 运算符的优先级。各运算符根据优先级从上到下降序排列。

运算符	顺序关联性	类型
()	从左到右	圆括号
**	从右到左	求幂
* / // %	从左到右	乘
+ -	从左到右	加
< <= > >=	从左到右	关系
== != <>	从左到右	相等
and	从左到右	逻辑 AND
or	从左到右	逻辑 OR
not	从右到左	逻辑 NOT

图 3.31 运算符的优先级和顺序关联性

3.17 结构化编程总结

建筑师设计建筑物时，必须综合运用历年积累的行业知识与经验。程序员设计程序时，也不例外。不过，这个行业要比建筑业年青，所以积累的知识与经验较少。通过以前的学习，我们知道同非结构化程序相比，结构化编程所生成的程序要容易理解得多，所以更易测试、调试和修改，而且不易出错。

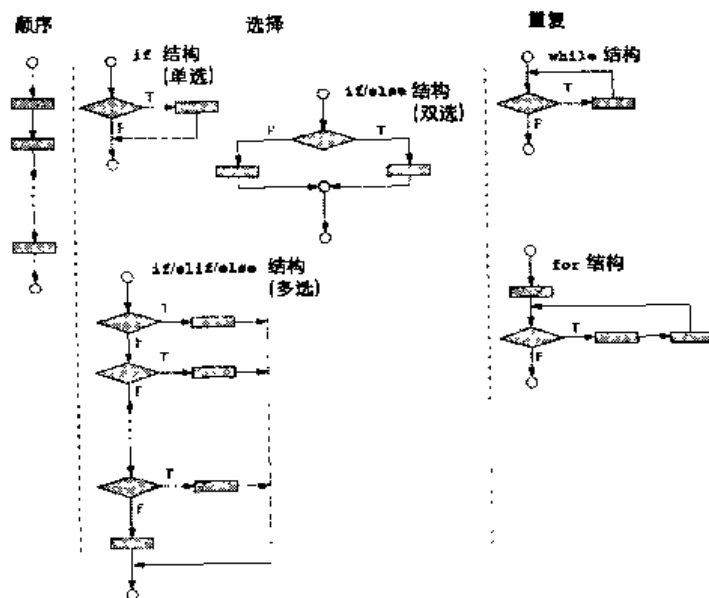


图 3.32 单入/单出的顺序、选择和重复结构

图 3.32 总结了 Python 控制结构。小圆圈表示每种结构的单一入口和出口。如果任意连接各种流程

图符号，可能会开发出非结构化的程序。因此，设计者决定合并流程图符号，构成一个有限的控制结构集，并只允许用两种简单方式来正确合并控制结构，以构建最终的结构化程序。

为简化起见，我们使用了单入 / 单出控制结构。对于每个控制结构，只有一条路可以进入，也有一条路可以退出。顺序连接不同控制结构，可方便地构建一个结构化程序——把一个控制结构的出口同下一个控制结构的入口连起来即可。也就是说，各控制结构在程序中一个接一个地放在一起——这称为“控制结构的堆叠”。结构化程序的构建规则也允许进行控制结构的嵌套。

图 3.33 总结了正确构建结构化程序的规则。规则假定用流程图中的矩形符号标注一项行动，包括输入和输出等等。规则还假定从最简流程图（图 3.34）开始。

结构化程序的构建规则：

- (1) 从如图 3.34 所示的“最简流程图”开始
- (2) 任何矩形（行动）都可被替换成两个顺序的矩形（行动）
- (3) 任何矩形（行动）都可被替换成任何控制结构（顺序、if、if/else、if/elif/else、while 或 for）
- (4) 规则（2）和（3）可根据需要应用任意次数，并按任意顺序应用

图 3.33 构建结构化程序的规则

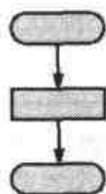


图 3.34 最简流程图

按图 3.33 的规则操作，肯定能获得一个条理清晰的、积木式的结构化流程图。例如，将规则（2）重复应用于最简流程图，最终的结构化流程图中会包含一系列顺序排列的矩形，如图 3.35 所示。注意由于规则（2）可生成控制结构的堆叠，所以该规则也称为“堆叠规则”。

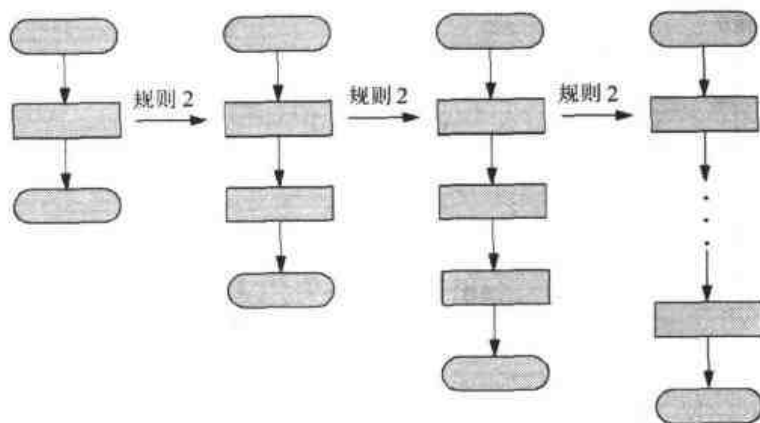


图 3.35 将图 3.33 的规则（2）应用于最简流程图

规则（3）称为“嵌套规则”。将规则（3）重复应用于最简流程图，会生成一个嵌套控制结构。例如在图 3.36 中，首先用一个双选（if/else）结构替换最简流程图中的矩形。随后，再次将规则（3）应用于双选结构中的两个矩形，用双选结构替换每个矩形。围绕每个双选结构的虚线框表示被替换的矩形。

规则（4）可生成更大的、头绪更多的、嵌套更深的结构。应用图 3.33 的规则，可获得任何可能的结构化流程图，所以也能生成任何可能的结构化程序。

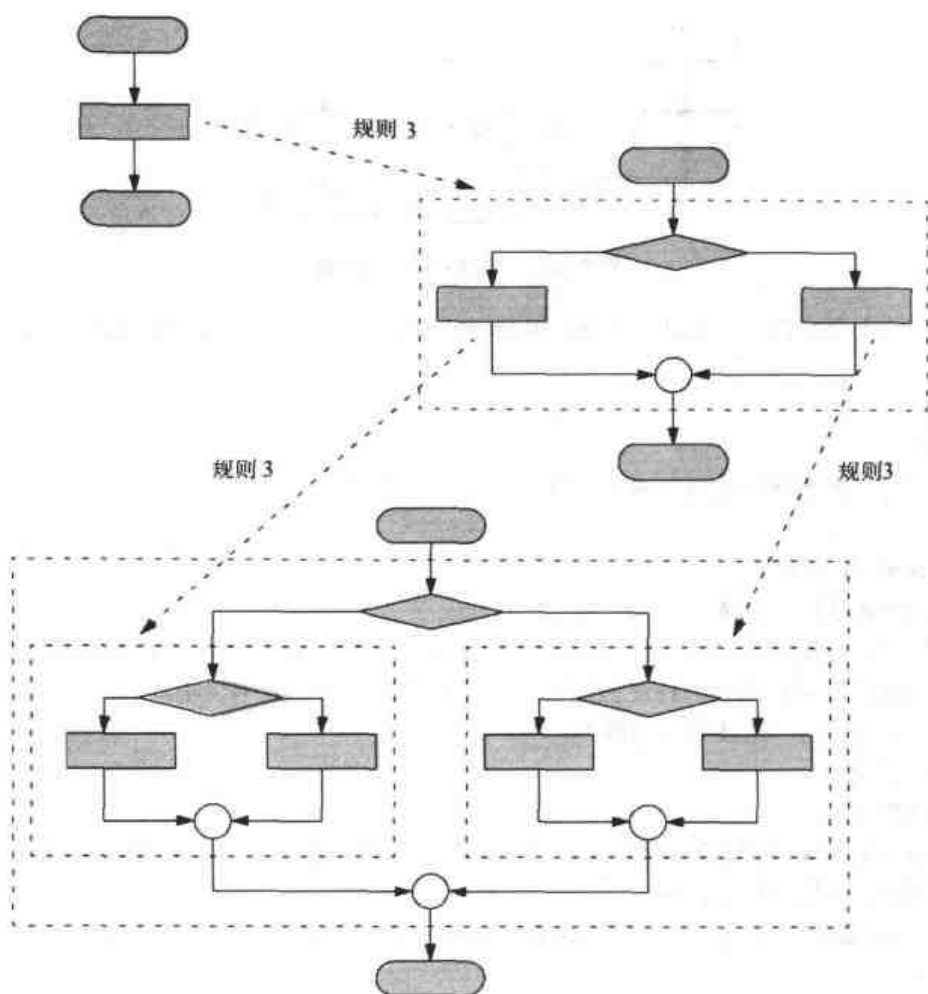


图 3.36 将图 3.35 的规则 (3) 应用于最简流程图

结构化编程的好处在于，整个过程只需使用 6 种简单的单入 / 单出结构，而且只需使用 2 种简单方法来“组装”它们。图 3.37 展示了一系列堆叠的“积木”，应用规则 (2) 生成；以及一系列嵌套的“积木”，应用规则 (3) 生成。该图还显示了积木重叠的情况，它们不可出现在结构化流程图中（因为已取消了 goto 语句）。

遵循图 3.33 的规则，不可能得到一个非结构化流程图（像图 3.38 那样）。如果不能确定流程图是否结构化，可反方向应用图 3.33 的规则，看看是否能将流程图简化成最简形式。如果能还原成最简流程图，就表明原来的流程图是结构化的；否则是非结构化的。

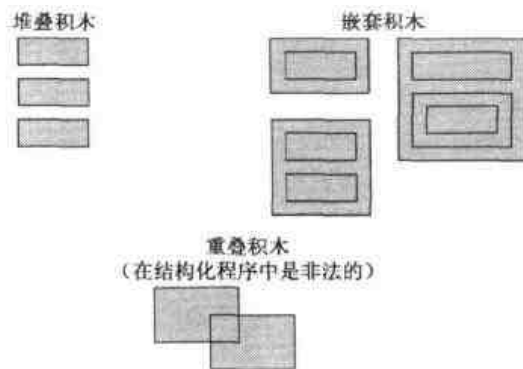


图 3.37 堆叠、嵌套和重叠积木

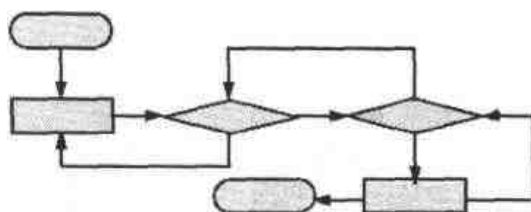


图 3.38 非结构化流程图

结构化编程的主要宗旨就是简化。Bohm 和 Jacopini 告诉我们，只需 3 种形式的控制：

- 顺序
- 选择
- 重复

顺序控制最简单。选择控制可采取以下 3 种方式之一实现：

- if 结构（单选）
- if/else 结构（双选）
- if/elif/else 结构（多选）

其实，很容易便可证明简单的 if 结构足以实现任何形式的选择。也就是说，用 if/else 和 if/elif/else 结构能做到的事情，通过合并 if 结构一样能做到（尽管可能条理不清，效率也不高）。

可用以下 3 种方式之一来实现重复控制：

- while 结构
- for 结构

同样很容易证明，仅用 while 结构便足以实现任何形式的重复。也就是说，用 for 结构能做到的事情，用 while 结构也能做到（尽管可能有点难度）。

综上所述，Python 程序需要的任何形式的控制都可通过下面 3 种形式来表示：

- 顺序
- if 结构（用于选择）
- while 结构（用于重复）

另外，这些控制结构只需选用两种方式（即堆叠和嵌套）来组合。总而言之，结构化编程大大简化了编程工作。

本章讨论了如何利用包含行动和决策的控制结构来构建程序。下一章要介绍另一种程序结构单元，即“函数”。您将学习如何通过合并各个函数（每个函数又可包含多种控制结构）来构建大型程序。此外，还要讨论函数是如何增强软件重用性的。在第 7 章，我们将介绍 Python 的另一种程序构建单元，即“类”。此后，还将用类来创建对象，开始进入面向对象编程（OOP）技术的学习。

第4章 函 数

学习目标

- 理解如何利用函数模块化地构建程序
- 会新建函数
- 理解函数间的信息传递机制
- 会利用随机数生成来实现模拟
- 理解标识符的可见性如何被限定在程序的特定区域
- 理解如何编写和使用递归函数（即能够调用自身的函数）
- 理解默认和关键字参数

4.1 概述

解决实际问题的大多数计算机程序都比前几章介绍的大。经验表明，开发和维护一个大程序时，最好是基于较小的部分或“组件”构建。同原始程序相比，每个组件都更易管理。这称为“分而治之”。本章将介绍用于简化大型程序设计、实现、运行和维护的许多 Python 语言特性。

4.2 Python 中的程序组件

Python 的程序组件包括函数、类、模块和包。程序员通常将自己定义的函数和类与现成 Python 模块提供的函数和类进行合并，从而写出新程序。“模块”（module）是包含函数和类定义的文件。许多模块可组合成一个集合，称为“包”（package）。本章重点在于函数，并适当介绍了模块和包；类的详情在第 7 章讲解。

程序员可定义函数以执行特定任务，这些任务可在程序的多个位置进行。这称为“程序员自定义函数”。定义函数的实际语句只需编写一次，但可在程序的多个位置调用。因此，函数是在 Python 中实现“软件重用”的一种基本单元，因其实现了程序代码的重用。

Python 模块提供了用于执行常见任务的函数，比如数学计算、字符串处理、字符处理、Web 编程、图形编程和其他许多操作。函数简化了程序员的工作，因为他们不必再写新函数来执行常见任务。模块集合（即“标准库”）被作为核心 Python 语言的一部分提供。这些模块位于 Python 安装目录的库目录下。在 Unix/Linux 上，是 `/usr/lib/python2.2` 或 `/usr/local/lib/python2.2`；在 Windows 上，则是 `\Python\Lib` 或 `\Python22\Lib`。

“模块”组合了相关定义，“包”则组合了相关模块。包作为一个整体，提供了帮助程序员完成常规任务的工具（比如图形或音频编程）。包中每个模块都定义了类、函数或数据，用于执行特定的、相关的任务（比如创建颜色、处理 .wav 文件等）。本书介绍了许多现成的 Python 包，如何创建一个可靠的包，则属于软件工程的范畴，不在本书进行讨论。

良好编程习惯 4.1 尽快熟悉核心 Python 模块提供的函数和类集合。

软件工程知识 4.1 避免重复别人的劳动。尽量使用标准库模块函数，不要写新函数。这可加快程序开发进度，并增强可靠性，因为您使用的是经过良好设计和测试的代码。

移植性提示 4.1 使用核心 Python 模块中的函数，通常可使程序更易移植。

性能提示 4.1 不要试图改写现成的模块函数使其更高效。这些函数已非常完美了。

函数需要被调用才能完成其目标任务。在函数调用中，需要指定函数名称，并提供函数执行任务所需的信息（比如参数）。这类似于公司的层次管理系统。老板（即“调用函数”或“调用者”）要求一名员工（即“被调用函数”）完成一项任务，并在完成后返回（汇报）结果。老板不关心员工具体如何完成任务。员工完全能调用其他员工，而老板根本注意不到。后面会解释如何通过“隐藏”实现细节来促进良好的软件工程。图 3.1 展示了老板函数（boss）函数如何通过一个层次化结构，与员工函数（worker1，worker2 和 worker3）通信。boss 在调用 worker1 时，不必知道 worker1 同 worker4 和 worker5 的关系。注意函数的实际关系可能有别于这张图显示的层次结构。

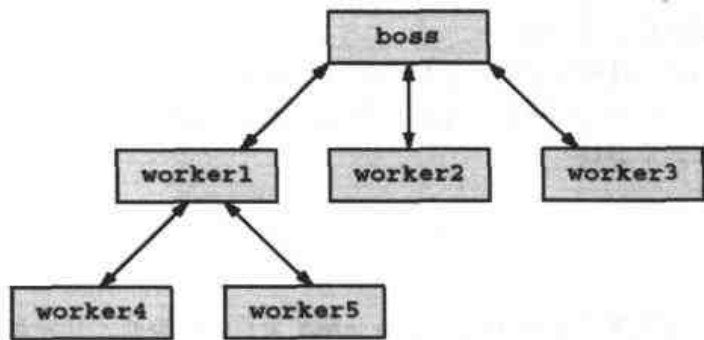


图 4.1 层次化的 boss/worker 函数关系

4.3 函数

程序员利用函数来模块化一个程序。函数定义中创建的所有变量都是“局部变量”——只存在于声明它们的函数中。许多函数都有一个参数列表，用于进行函数间的通信，这些参数也是局部变量。

有几方面的动机要求对一个程序进行“函数化”。首先，“分而治之”技术使程序开发更易管理。其次是“软件重用性”——以现有函数为基础来创建新程序。软件重用性是面向对象编程的一个主要优势（详情参见第 7 章、第 8 章和第 9 章）。采用良好的函数命名和定义，程序可通过负责特定任务的标准化函数来创建，而不必为每个任务都编写自定义的代码。最后则是在程序中避免重复代码。把代码包装成一个函数，可通过调用函数在不同地方执行这些代码，而不必在用到它们的每个地方都重写一遍。

软件工程知识 4.2 每个函数都应该只限执行单一的、良好定义的任务，函数名应清楚地描述那个任务。这样有利于提升软件的重用性。

软件工程知识 4.3 如实在想不出能准确表达函数作用的名称，就表明函数可能执行了太多分散的任务。通常，最好将这种函数分解成多个更小的函数。

4.4 math 模块的函数

模块包含了函数定义和执行相关任务的其他元素（例如类定义）。利用 math 模块的函数，可执行特定的数学计算。我们用 math 模块的各个函数来介绍函数和模块的概念。书中各处还将讨论核心 Python 模块中的其他许多函数。

调用一个函数时，通常要写下它的名称，后续一个左圆括号、要传给函数的参数（或者用逗号分隔的参数值列表）以及一个右圆括号。要使用模块定义的一个函数，程序必须导入模块，这是用关键字 import 来实现的。模块导入后，程序即可调用模块中的函数，这需要使用模块名、一个小圆点（.）和正式的函数调用，即 `moduleName.functionName()`。图 4.2 的交互式会话演示了如何用 math 模块打印 900 的平方根。

```

Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> print math.sqrt( 900 )
30.0
>>> print math.sqrt( -900 )
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ValueError: math domain error

```

图 4.2 math 模块的 sqrt 函数

执行下面这一行时:

```
print math.sqrt( 900 )
```

math 模块的 sqrt 函数会计算括号中包含的数字(900)的平方根。900 是 math.sqrt 函数的参数。函数会返回浮点值 30.0, 以便在屏幕上显示。执行下面这一行时

```
print math.sqrt( -900 )
```

函数调用会生成一个错误, 即“异常”, 因为 sqrt 函数不能处理负的参数值。解释器将在屏幕上显示同这个错误有关的信息。异常和异常处理的问题将在第 12 章讨论。

常见编程错误 4.1 使用 math 模块的函数时, 忘记导入 math 模块属于运行时错误。只有导入模块, 才能使用其中的函数和变量。

常见编程错误 4.2 用 import 语句导入模块后, 如果忘记为它的函数附加模块名前缀, 就会造成运行时错误。

函数的参数可为值、变量或表达式。假定 $c1 = 13.0$, $d = 3.0$, 而 $f = 4.0$, 那么下述语句:

```
print math.sqrt( c1 - d * f )
```

会计算并打印 $13.0 - 3.0 * 4.0 = 25.0$ 的平方根, 结果为 5.0。图 4.3 总结了其他一些 math 模块函数, 注意有的结果已经取整。

函数	说明	示例
<code>acos(x)</code>	求 x 的反余弦 (结果是弧度)	<code>acos(1.0)</code> 等于 0.0
<code>asin(x)</code>	求 x 的反正弦 (结果是弧度)	<code>asin(0.0)</code> 等于 0.0
<code>atan(x)</code>	求 x 的反正切 (结果是弧度)	<code>atan(0.0)</code> 等于 0.0
<code>ceil(x)</code>	为 x 取整, 结果是不小于 x 的最小 的整数	<code>ceil(9.2)</code> 等于 10.0 <code>ceil(-9.8)</code> 等于 -9.0
<code>cos(x)</code>	求 x 的余弦 (x 是弧度)	<code>cos(0.0)</code> 等于 1.0
<code>exp(x)</code>	求幂函数 e^x	<code>exp(1.0)</code> 等于 2.71828 <code>exp(2.0)</code> 等于 7.38906
<code>fabs(x)</code>	求 x 的绝对值	<code>fabs(5.1)</code> 等于 5.1 <code>fabs(-5.1)</code> 等于 5.1
<code>floor(x)</code>	为 x 取整, 结果是不大于 x 的最大 的整数	<code>floor(9.2)</code> 等于 9.0 <code>floor(-9.8)</code> 等于 -10.0
<code>fmod(x, y)</code>	求 x/y 的余数, 结果是浮点数	<code>fmod(9.8, 4.0)</code> 等于 1.8
<code>hypot(x, y)</code>	求直角三角形的斜边长度, 直边长 度为 x 和 y : $\sqrt{x^2 + y^2}$	<code>hypot(3.0, 4.0)</code> 等于 5.0

函数	说明	示例
<code>log10(x)</code>	求 x 的对数 (以 10 底)	<code>log10(10.0)</code> 等于 1.0 <code>log10(100.0)</code> 等于 2.0
<code>pow(x, y)</code>	求 x 的 y 次方 (x^y)	<code>pow(2.0, 7.0)</code> 等于 128.0 <code>pow(9.0, .5)</code> 等于 3.0
<code>sin(x)</code>	求 x 的正弦 (x 是弧度)	<code>sin(0.0)</code> 等于 0.0
<code>sqrt(x)</code>	求 x 的平方根	<code>sqrt(900.0)</code> 等于 30.0 <code>sqrt(9.0)</code> 等于 3.0
<code>tan(x)</code>	求 x 的正切 (x 是弧度)	<code>tan(0.0)</code> 等于 0.0

图 4.3 math 模块函数

4.5 函数定义

前面所展示的每个程序都由一系列语句构成，它们称为“预定义 Python 函数”，用于完成程序的任务。我们将这些语句称为“程序的主要部分”，以将它们与包含函数定义的其余部分区分开。下面要讨论如何编写自定义函数。

软件工程知识 4.4 在包含许多函数的程序中，主要部分应该是对函数的一系列调用，再由那些函数实际执行程序的大量工作。

假定在一个程序里使用了用户自定义函数 `square`，它计算从 1~10 的所有整数的平方，如图 4.4 所示。函数在使用之前，必须先定义。

良好编程习惯 4.2 在函数定义之间放入一个空行，以便区分函数，并增强程序的可读性。

```

1 # Fig. 4.4: fig04_04.py
2 # Creating and using a programmer-defined function.
3
4 # function definition
5 def square( y ):
6     return y * y
7
8 for x in range( 1, 11 ):
9     print square( x ),
10
11 print

```

```
1 4 9 16 25 36 49 64 81 100
```

图 4.4 程序员自定义函数

主程序的第 9 行通过以下语句调用函数 `square` (在第 5~6 行定义):

```
print square( x ),
```

这样，函数 `square` 会在参数 y 中收到 x 的一个拷贝。^①然后，`square` 计算 $y * y$ (第 6 行)。结果返回给调用 `square` 的语句。函数调用 (第 9 行) 求值为由函数的返回值，并由 `print` 语句显示。 x 值不会被函数调用更改。使用 `for` 重复结构，这个过程要重复 10 次。函数定义的形式是：

```

def function-name( parameter-list ):
    statements

```

^① y 实际收到的是对 x 的一个引用，但 y 将其视为 x 值的一个拷贝。这是“传递对象引用”的概念，详情参见第 15 章。

其中, *function-name* (函数名) 是任何有效标识符, *parameter-list* (参数列表) 是用逗号分隔的参数名列表, 这些参数可由 *function-name* 接收。如函数不接收任何值, 参数列表将是空白的, 只是仍需圆括号。`def` 语句之后缩进的语句构成了“函数主体”, 称为 *block* (块或代码块)。

常见编程错误 4.3 忘记在函数参数列表后添加冒号 (:) 是语法错误。

常见编程错误 4.4 函数调用中的圆括号是 Python 运算符, 是它导致了函数调用。如遗漏圆括号, 函数不会被调用。通常, 控制权会转交给语句。如 `print` 语句包含一个无圆括号的函数调用, 会显示函数的内存位置。如用户希望将函数调用的结果指派给函数, 没有圆括号的函数调用就会将函数本身同变量绑定。

常见编程错误 4.5 函数主体没有缩进是语法错误。

良好编程习惯 4.3 不建议为传给函数的参数和函数定义中的对应参数使用相同的名称。

良好编程习惯 4.4 选择有意义的函数名和参数名有利于增强程序可读性, 还可减少注释量。这样写程序可获得“自文档化的代码”。

软件工程知识 4.5 函数尽量不要超过编辑器窗口的宽度。不管函数有多长, 都应很好地执行一项任务。小函数有利于提升软件的重用性。

测试和调试提示 4.1 和在整个程序中更新重复的代码相比, 更新函数显然更容易。

软件工程知识 4.6 程序应该写为若干小函数的集合。这使程序更易编写、调试、维护和修改。

软件工程知识 4.7 如函数需要大量参数, 表明它执行的任务可能过多。请考虑将函数分解成更小的函数, 令其执行单独的任务。函数的 `def` 语句尽可能不超过一行。

函数任务完成后, 会将控制权返回给调用者。有 3 种方式能将控制权返回函数的调用位置。如函数不显式返回一个结果, 一旦抵达最后一个缩进的行, 或者执行到 `return` 语句, 控制权就会返回。在这两种情况下, 函数返回的都是 `None`, 它是 Python 用于表示“空”的一个值, 即没有声明任何值。此外, 在条件表达式中, 它将求值为 `false`。

如函数要返回结果, 以下语句:

```
return expression
```

会将 *expression* (表达式) 的值返回给调用者。

第二个例子 (图 4.5) 使用了程序员自定义函数 `maximumValue`。该函数独立于其参数的类型。我们用 `maximumValue` 函数判断并返回 3 个整数、3 个浮点数以及 3 个字符串中的最大值。

```
1 # Fig. 4.5: fig04_05.py
2 # Finding the maximum of three integers.
3
4 def maximumValue( x, y, z ):
5     maximum = x
6
7     if y > maximum:
8         maximum = y
9
10    if z > maximum:
11        maximum = z
12
13    return maximum
14
15 a = int( raw_input( "Enter first integer: " ) )
16 b = int( raw_input( "Enter second integer: " ) )
17 c = int( raw_input( "Enter third integer: " ) )
18
19 # function call
```



```

20 print "Maximum integer is:", maximumValue( a, b, c )
21 print # print new line
22
23 d = float( raw_input( "Enter first float: " ) )
24 e = float( raw_input( "Enter second float: " ) )
25 f = float( raw_input( "Enter third float: " ) )
26 print "Maximum float is: ", maximumValue( d, e, f )
27 print
28
29 g = raw_input( "Enter first string: " )
30 h = raw_input( "Enter second string: " )
31 i = raw_input( "Enter third string: " )
32 print "Maximum string is: ", maximumValue( g, h, i )

```

```

Enter first integer: 27
Enter second integer: 12
Enter third integer: 36
Maximum integer is: 36

Enter first float: 12.3
Enter second float: 45.6
Enter third float: 9.03
Maximum float is: 45.6

Enter first string: hello
Enter second string: programming
Enter third string: goodbye
Maximum string is: programming

```

图 4.5 程序员自定义函数 maximumValue

第 15 行将两个函数调用 (raw_input 和 int) 合为一个语句。在本例中, 函数 raw_input 从用户输入中读取一个值, 再将结果作为参数传给函数 int。对函数 maximumValue 的调用 (第 20 行) 将 3 个整数传递给程序员定义的函数 (第 4~13 行)。maximumValue 中的 return 语句 (第 13 行) 将最大值返回主程序。print 语句 (第 20 行) 显示返回值。同样的函数还会返回最大浮点值 (第 26 行) 和最大字符串 (第 32 行)。

4.6 随机数生成

下面将简要探讨一种流行的编程应用——模拟和博弈, 我们展示了到目前为止学过的大多数控制结构。本节和下一节将开发一个游戏程序, 它集成了多个函数。

赌博是人的“天性”。在美国, 人们涌向各种赌博场所, 以各种方式挥散金钱。赌博是一种典型“投机”行为, 幸运儿可在短时间内把一口袋钱变成一座金山。在 Python 程序中, 要用 random 模块来模拟“投机”。

函数 random.randrange 可生成一个随机整数, 范围从第一个参数值开始, 一直到 (但不包括) 第二个参数值。假如 randrange 真的能随机地生成整数, 那么每次调用函数时, 范围中每个数字出现的概率都应该是均等的。

图 4.6 显示了模拟掷 20 次骰子的结果 (骰子共有 6 面), 以演示 random 模块的用法。random.randrange(1, 7) 能生成 1~6 之间的整数。

```

1 # Fig. 4.6: fig04_06.py
2 # Random integers produced by randrange.
3
4 import random
5
6 for i in range( 1, 21 ): # simulates 20 die rolls
7     print "%10d" % ( random.randrange( 1, 7 ) ),
8
9     if i % 5 == 0: # print newline every 5 rolls
10         print

```

5	3	3	3	2
3	2	3	3	4
2	3	6	5	4
6	2	4	1	2

图 4.6 用 `random.randrange(1, 7)` 生成随机数

为证明数字出现机会均等, 下面模拟掷 6000 次骰子 (图 4.7)。1~6 的每个整数都应出现约 1000 次。

```

1 # Fig. 4.7: fig04_07.py
2 # Roll a six-sided die 6000 times.
3
4 import random
5
6 frequency1 = 0
7 frequency2 = 0
8 frequency3 = 0
9 frequency4 = 0
10 frequency5 = 0
11 frequency6 = 0
12
13 for roll in range(1, 6001):      # 6000 die rolls
14     face = random.randrange(1, 7)
15
16     if face == 1:                # frequency counted
17         frequency1 += 1
18     elif face == 2:
19         frequency2 += 1
20     elif face == 3:
21         frequency3 += 1
22     elif face == 4:
23         frequency4 += 1
24     elif face == 5:
25         frequency5 += 1
26     elif face == 6:
27         frequency6 += 1
28     else:                        # simple error handling
29         print "should never get here!"
30
31 print "Face %13s" % "Frequency"
32 print "  1 %13d" % frequency1
33 print "  2 %13d" % frequency2
34 print "  3 %13d" % frequency3
35 print "  4 %13d" % frequency4
36 print "  5 %13d" % frequency5
37 print "  6 %13d" % frequency6

```

Face	Frequency
1	946
2	1003
3	1035
4	1012
5	987
6	1017

图 4.7 掷 6000 次骰子

如输出所示, 函数 `random.randrange` 可成功模拟掷一个 6 面骰子的情况。注意, 在 `if/elif/else` 结构中, 程序永远不会执行 `else` 条件 (第 28~29 行), 但我们仍然添加了这个条件, 以强调良好的编程习惯。

测试和调试提示 4.2 即使绝对肯定程序没有 bug, 也应在 `if/elif/else` 结构中添加一个默认 `else` 条件。

4.7 示例: 博彩游戏

目前流行的博彩游戏中, 有一种名为“掷双骰”(Craps)的游戏, 规则非常简单:

玩家掷两粒骰子。每粒骰子都有 6 面，分别标上 1, 2, 3, 4, 5 和 6 这几个点。骰子停止滚动之后，计算它们朝上那一面的点数之和。如果首次掷出后点数之和为 7 或 11，那么玩家赢（庄家输）。首次掷出后点数之和为 2, 3 或者 12，玩家输（庄家赢）。首次掷出后点数之和为 4, 5, 6, 8, 9 或者 10，这些数字会成为玩家的“目标点”（Point）。要想赢，玩家必须不断掷骰子，直到点数与目标点相同。但之前如果掷出一个 7 点，会马上输掉本局。

图 4.8 的程序模拟了掷双骰游戏，并显示了 4 次示范执行结果。

```

1 # Fig. 4.8: fig04_08.py
2 # Craps.
3
4 import random
5
6 def rollDice():
7     die1 = random.randrange( 1, 7 )
8     die2 = random.randrange( 1, 7 )
9     workSum = die1 + die2
10    print "Player rolled %d + %d = %d" % ( die1, die2, workSum )
11
12    return workSum
13
14 sum = rollDice()                # first dice roll
15
16 if sum == 7 or sum == 11:       # win on first roll
17     gameStatus = "WON"
18 elif sum == 2 or sum == 3 or sum == 12: # lose on first roll
19     gameStatus = "LOST"
20 else:                           # remember point
21     gameStatus = "CONTINUE"
22     myPoint = sum
23     print "Point is", myPoint
24
25 while gameStatus == "CONTINUE": # keep rolling
26     sum = rollDice()
27
28     if sum == myPoint:           # win by making point
29         gameStatus = "WON"
30     elif sum == 7:               # lose by rolling 7:
31         gameStatus = "LOST"
32
33 if gameStatus == "WON":
34     print "Player wins"
35 else:
36     print "Player loses"

```

```

Player rolled 2 + 5 = 7
Player wins

```

```

Player rolled 1 + 2 = 3
Player loses

```

```

Player rolled 1 + 5 = 6
Point is 6
Player rolled 1 + 6 = 7
Player loses

```

```

Player rolled 5 + 4 = 9
Point is 9
Player rolled 4 + 4 = 8
Player rolled 2 + 3 = 5
Player rolled 5 + 4 = 9
Player wins

```

图 4.8 掷双骰游戏

注意，每次都要掷两粒骰子。rollDice 函数模拟掷骰子（第 6~12 行）。rollDice 只定义了一次，但

在程序的两个地方都调用了它（第 14 行和第 26 行）。函数不需要参数，所以参数列表是空的。rollDice 函数最后打印并返回两粒骰子的和（第 10~12 行）。

这个游戏非常好玩。第一次和以后任何一次掷骰子时，玩家都有可能赢，也有可能输。gameStatus 变量跟踪输赢状态。gameStatus 变量可能是 3 个字符串之一：“WON”，“LOST”或者“CONTINUE”。如果取胜，gameStatus 会设为“WON”（第 17 和第 29 行）。如果输了，gameStatus 会设为“LOST”（第 19 行和第 31 行）。否则，gameStatus 会设为“CONTINUE”，玩家可以继续掷骰子。

首次掷骰子时，无论输赢，while 结构主体（第 25~31 行）都会跳过，因为 gameStatus 不等于“CONTINUE”。相反，程序会继续 if/else 结构（第 33~36 行）。如果 gameStatus 等于“WON”，将打印“Player wins”；如果 gameStatus 等于“LOST”，将打印“Player loses”。

首次掷骰子时，如果既不赢，也不输，sum 的值会被指派给变量 myPoint（第 22 行）。程序要进入 while 结构，因为 gameStatus 等于“CONTINUE”。每次执行 while 循环，都会调用 rollDice 来生成一个新的 sum（第 26 行）。如 sum 与 myPoint 相符，gameStatus 会设为“WON”（第 28~29 行）。与此同时，while 测试会失败（第 25 行），而 if/else 结构要打印“Player wins”（第 33~34 行），执行终止。如 sum 等于 7，gameStatus 会设为“LOST”（第 30~31 行），while 测试同样会失败，而 if/else 结构要打印“Player loses”（第 35~36 行），执行终止。否则，会继续 while 循环。

注意，这里运用了以前讨论过的各种程序控制机制。掷双骰游戏使用了一个程序员自定义函数 rollDice，以及 while，if/else 和 if/elif/else 结构。示例程序中既使用了堆叠控制结构（第 16~23 行的 if/elif/else，以及第 25~31 行的 while），也使用了嵌套控制结构（第 28~31 行的 if/elif，它嵌套在第 25~31 行的 while 内）。

4.8 作用域规则^①

到目前为止，我们一直没有讨论 Python 程序怎样存储和获取变量值。从表面看，当程序需要时，似乎值就“摆在那里”。但事实上，Python 严格规定了什么时候以什么方式访问一个变量的值。这些规则是通过“命名空间”和“作用域”来描述的。本节探讨了命名空间和作用域怎样影响程序的执行。

下面用一个例子来解释这些概念。假定一个函数包含下述代码行：

```
print x
```

在值能够打印到屏幕之前，Python 首先必须找到名为 x 的标识符，并判断同该标识符对应的值。命名空间存储着与标识符及其绑定的值有关的信息。Python 定义了 3 个命名空间，分别是局部（local）、全局（global）和内建（built-in）命名空间。程序试图访问标识符的值时，Python 会按特定顺序搜索命名空间（即局部、全局和内建命名空间），检查是否存在标识符，以及它的具体位置。

Python 搜索的第一个命名空间是局部命名空间，它存储着一个代码块（block）中创建的绑定。函数主体是 block，所以所有函数参数以及函数创建的任何标识符都存储在函数的局部命名空间。每个函数都有一个惟一的局部命名空间，一个函数不能访问另一个函数的命名空间。在上例中，Python 首先搜索函数的局部命名空间，查找一个名为 x 的标识符。如函数的局部命名空间包含这样的标识符，函数会在屏幕上打印 x 的值。如函数的局部命名空间没有名为 x 的标识符（即函数没有定义任何参数，也没有任何创建名为 x 的标识符）。Python 会搜索下一个外层命名空间，即全局命名空间（有时也称为模块命名空间）。

全局命名空间包含定义于一个模块或文件中的所有标识符、函数名和类名的绑定。每个模块或文件的全局命名空间都包含名为 __name__ 的一个标识符，它指出了模块名（例如“math”或“random”）。一旦启动 Python 解释器会话，或 Python 解释器开始执行存储在文件中的一个程序，__name__ 的值是“__main__”。

^① 本书附录 B 简要讨论了嵌套作用域。这是一个复杂的主题，它在 Python 2.1 中是选择使用，在 Python 2.2 中则是强制使用。与嵌套作用域有关的更多信息可在 PEP 227 中找到，地址是 www.python.org/peps/pep-0227.html。

在上例中, Python 在全局命名空间搜索名为 `x` 的一个标识符。如全局命名空间包含该标识符(即标识符在函数调用之前绑定到全局命名空间), Python 会停止搜索标识符, 函数会将 `x` 的值打印到屏幕。如全局命名空间不包含名为 `x` 的标识符, Python 会搜索下一个外层命名空间(即内建命名空间)。

内建命名空间包含了对应于许多 Python 函数和错误消息的标识符。例如, 函数 `raw_input`、`int` 和 `range` 都从属于内建命名空间。Python 会在解释器启动时创建内建命名空间, 而且程序通常不会修改命名空间(比如为命名空间添加一个标识符等)。在上例中, 内建命名空间不包含名为 `x` 的标识符, Python 停止搜索, 打印错误消息, 指出未找到标识符。

标识符的“作用域”描述了程序的什么区域可访问标识符的值。如标识符定义于局部命名空间(即函数), `block` 中的所有语句都可访问该标识符。`block` 之外的语句(比如在程序的主要部分, 或者在另一个函数中)则不能访问标识符。一旦代码块终止(比如在 `return` 语句之后), `block` 中所有标识符的局部命名空间都会“超出作用域”, 变得不可访问。

如果标识符定义于全局命名空间, 标识符就具有了“全局作用域”。从标识符创建位置开始, 一直到文件结束, 其间执行的所有代码都能访问全局标识符。此外, 如满足特定条件, 函数还可访问全局标识符。稍后还会详细讨论这个问题。包含在内建命名空间中的标识符也许能由程序、模块或函数中的代码访问。

在使用了函数的程序中, 要注意避免“遮蔽”(Shadowing)问题。如函数创建的局部标识符与模块或内建命名空间中的标识符同名, 局部标识符就会“遮蔽”全局或内建标识符。如果程序表面引用局部变量, 但真正想引用的却是全局或内建标识符, 就可能引起逻辑错误。

常见编程错误 4.6 用局部命名空间中的一个标识符遮蔽模块或内建命名空间中的一个标识符, 可能引起逻辑错误。

良好编程习惯 4.5 避免变量名遮蔽外层作用域中的名称。为此, 要注意避免标识符与内建命名空间中的标识符同名, 并避免在程序中使用重复的标识符。

Python 允许程序员查看当前命名空间提供了哪些标识符。内建函数 `dir` 会返回这些标识符的一个列表。图 4.9 展示了在启动一个交互式会话时, 由 Python 创建的命名空间。调用函数 `dir` 后, 我们知道当前命名空间包含 3 个标识符: `__builtins__`、`__doc__` 和 `__name__`。下一个命令打印标识符 `__name__` 的值, 以便向交互式会话说明, 它的值是 `__main__`。后续的命令打印标识符 `__builtins__` 的值。结果表明, 该标识符与一个模块绑定在一起。所以能通过标识符 `__builtins__` 来引用模块 `__builtin__`。这方面的详情将在 4.9 节介绍。下一个命令创建新标识符 `x`, 并把它与值 3 绑定在一起。再次调用函数 `dir`, 证明标识符 `x` 已成功添加到会话的命名空间。

图 4.9 中的交互式会话只说明了这一点: Python 程序能够充分提供有关程序(或交互式会话)中标识符的相关信息, 这称为“内省”。Python 提供了其他许多内省能力, 其中包括函数 `globals` 和 `locals`。这两个函数分别返回全局命名空间和局部命名空间。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> print __name__
__main__
>>> print __builtins__
<module '__builtin__' (built-in)>
>>> x = 3 # bind new identifier to global namespace
>>> dir()
['__builtins__', '__doc__', '__name__', 'x']
```

图 4.9 函数 `dir`

尽管函数有利于程序的调试, 但假如开发者不小心, 作用域问题可能在程序中造成不易察觉的错误。

图 4.10 的程序演示了这些问题，它使用了全局和局部变量。第 4 行创建变量 `x`，并为其指派值 1。变量存在于程序的全局命名空间，并具有全局作用域。换言之，变量 `x` 可由第 4 行之后的任何代码访问和修改。该全局变量在任何创建了一个名为 `x` 的局部变量的函数中，都会被“遮蔽”。在主程序中，第 22 行打印变量 `x` 的值（即 1）。第 24~25 行将值 7 指派给变量 `x`，并打印新值。

```

1 # Fig. 4.10: fig04_10.py
2 # Scoping example.
3
4 x = 1 # global variable
5
6 # alters the local variable x, shadows the global variable
7 def a():
8     x = 25
9
10    print "\nlocal x in a is", x, "after entering a"
11    x += 1
12    print "local x in a is", x, "before exiting a"
13
14 # alters the global variable x
15 def b():
16     global x
17
18    print "\nglobal x is", x, "on entering b"
19    x *= 10
20    print "global x is", x, "on exiting b"
21
22 print "global x is", x
23
24 x = 7
25 print "global x is", x
26
27 a()
28 b()
29 a()
30 b()
31
32 print "\nglobal x is", x

```

```

global x is 1
global x is 7

local x in a is 25 after entering a
local x in a is 26 before exiting a

global x is 7 on entering b
global x is 70 on exiting b

local x in a is 25 after entering a
local x in a is 26 before exiting a

global x is 70 on entering b
global x is 700 on exiting b

global x is 700

```

图 4.10 作用域和 `global` 关键字

程序定义了两个函数，它们不接收或返回任何参数。函数 `a`（第 7~12 行）声明局部变量 `x`，并初始化为 25。然后，函数 `a` 打印局部变量 `x`，使其自增，再次打印它（第 10~12 行）。程序每次调用函数时，函数 `a` 都会重新创建局部变量 `x`，并将变量初始化成 25，再自增为 26。

函数 `b`（第 15~20 行）不声明任何变量。相反，第 16 行使用关键字 `global`，指定 `x` 具有全局作用域。因此，当函数 `b` 引用变量 `x` 时，Python 会在全局命名空间找到标识符 `x`。程序首次调用函数 `b` 时（第 28 行），会打开全局变量的值（7），将那个值乘以 10，再次打印全局变量的值（70），并退出函数。程序第二次调用函数 `b` 时（第 30 行），全局变量包含了修改过的值（70）。最后，第 32 行在主程序中再次打

印全局变量 `x (700)`，证明函数 `b` 已修改了该变量的值。

4.9 关键字 `import` 和命名空间

前面讨论了如何导入模块，以及如何使用那个模块中定义的函数。本节将探讨导入一个模块会对程序的命名空间带来哪些影响。此外还将介绍将模块导入程序的多种方式。

4.9.1 导入一个或多个模块

假定程序需要执行在 `math` 模块中定义的某个特定算术运算。程序首先必须导入模块：

```
import math
```

这样，导入模块的代码在其命名空间中，就有了一个对 `math` 模块的引用。在 `import` 语句之后，程序可访问 `math` 模块中定义的任何标识符。图 4.11 的交互式会话说明了 `import` 语句是如何影响会话的命名空间，以及程序是如何访问模块命名空间中定义的标识符的。第一行导入 `math` 模块。下一行调用函数 `dir`，证明标识符 `math` 已插入会话的命名空间。如后续的 `print` 语句所示，标识符同代表 `math` 模块的一个对象绑定。如果将标识符 `math` 传递给函数 `dir`，函数会返回 `math` 模块的命名空间中的所有标识符。^①注意，早期版本的 Python 可能为 `dir()` 输出不同的结果。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> dir()
['_builtins_', '__doc__', '__name__', 'math']
>>> print math
<module 'math' (built-in)>
>>> dir( math )
['_doc__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'sin',
'sinh', 'sqrt', 'tan', 'tanh']
>>> math.sqrt( 9.0 )
3.0
```

图 4.11 导入一个模块

会话中的下一个命令调用函数 `sqrt`。为访问 `math` 模块命名空间中的一个标识符，必须使用点访问运算符 (`.`)。下面这行：

```
math.sqrt( 9.0 )
```

首先访问 `math` 模块命名空间中定义的 `sqrt` 函数（通过点访问运算符），然后调用 `sqrt` 函数，并传递参数值 `9.0`（通过圆括号运算符）。如果程序要导入几个模块，就可为每个模块单独使用 `import` 语句。还可用一个语句导入多个模块（用逗号隔开模块名）。导入的每个模块都添加到程序的命名空间，参见图 4.12。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import math, random
>>> dir()
['_builtins_', '__doc__', '__name__', 'math', 'random']
```

图 4.12 导入多个模块

^① 函数 `dir` 实际返回的是作为参数传递的一个对象的属性列表。在模块的情况下，信息汇总成模块中定义的所有标识符（例如函数和数据）的一个列表。

4.9.2 从模块导入标识符

前面的例子讨论了如何访问在另一个模块的命名空间中定义的标识符。为访问标识符，必须使用点访问运算符 (.)。有时，程序只使用模块中的一个或几个标识符。在这种情况下，可以只导入程序需要的标识符。Python 提供了 `from/import` 语句，可将一个或多个标识符从模块直接导入程序的命名空间。

图 4.13 的交互式会话将 `sqrt` 函数直接导入会话的命名空间。解释器执行到下面这行时：

```
from math import sqrt
```

会创建一个对函数 `math.sqrt` 的引用，并将引用放入会话的命名空间。现在可直接调用函数，不必用点运算符。程序可在一个语句中导入多个模块，同样地，也可在一个语句中从模块导入多个标识符。如：

```
from math import sin, cos, tan
```

会将 `math` 的 `sin`、`cos` 和 `tan` 函数直接导入会话的命名空间。在 `import` 语句之后，对函数 `dir` 的调用揭示了所有这些函数。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from math import sqrt
>>> dir()
['_builtins_', '__doc__', '__name__', 'sqrt']
>>> sqrt(9.0)
3.0
>>> from math import sin, cos, tan
>>> dir()
['_builtins_', '__doc__', '__name__', 'cos', 'sin', 'sqrt', 'tan']
```

图 4.13 从模块导入标识符

图 4.14 的交互式会话说明了程序还能导入模块中定义的所有标识符。以下语句：

```
from math import *
```

能够从 `math` 模块将所有没有以下划线开头的标识符导入交互式会话的命名空间。现在，程序员能够调用 `math` 模块中的任何函数，同时不必通过点访问运算符来访问函数。然而，像这样导入模块的标识符可能导致严重错误，并被认为是一种危险的编程做法。假定程序定义了一个名为 `e` 的标识符，并将字符串值 `"e"` 指派给它。执行了上述 `import` 语句后，标识符 `e` 会与数学浮点常量 `e` 绑定到一起，以前的 `e` 值将无法访问。通常，程序永远都不要像这样从模块导入所有标识符。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from math import *
>>> dir()
['_builtins_', '__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh',
'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi',
'pow', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
```

图 4.14 从模块导入所有标识符

测试和调试提示 4.3 通常应避免将所有标识符从一个模块导入另一个模块的命名空间。只有可信来源提供的模块，才可考虑这种导入方法。在可信模块的文档中，应清楚说明可用这样的语句导入模块。

4.9.3 为模块和模块标识符绑定名称

前面讨论了如何导入模块，以及如何从模块导入特定标识符。Python 的语法规则允许程序员控制

import 语句影响程序命名空间的方式。本节更详细地讨论了这种控制，并进一步解释了程序员如何自定义对已导入的元素的引用。

以下语句：

```
import random
```

会导入 random 模块，并在命名空间中放入一个对 random 模块的引用。图 4.15 中交互式会话的以下语句：

```
import random as randomModule
```

也会导入 random 模块，但 as 子句允许程序员指定模块的引用名称。在本例中，我们创建了一个引用，名为 randomModule。以后要访问 random 模块，可直接使用 randomModule 引用

程序还可使用 import/as 语句为从模块导入的一个标识符指定名称。以下语句：

```
from math import sqrt as squareRoot
```

会从 math 模块导入 sqrt 函数，并创建对该函数的引用 squareRoot。以后可用该引用来调用函数。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import random as randomModule
>>> dir()
['_builtins_', '__doc__', '__name__', 'randomModule']
>>> randomModule.randrange(1, 7)
1
>>> from math import sqrt as squareRoot
>>> dir()
['_builtins_', '__doc__', '__name__', 'randomModule', 'squareRoot']
>>> squareRoot(9.0)
3.0
```

图 4.15 为导入的元素指定名称

模块作者经常使用 import/as 语句，因为对导入的元素来说，它所定义的名称可能与作者模块定义的标识符冲突。使用 import/as 语句，模块作者可为导入的元素指定新名称，以防止名称冲突。程序员也用 import/as 语句来简化操作。由于可为标识符指定一个较短的名称，所以能改善可读性，同时减少打字量。

Python 的元素导入功能有利于基于组件的编程。程序员应根据具体情况选择 Python 语法，注意，基于组件的编程的目标是创建更容易构建和维护的程序。

4.10 递归

以前讨论的程序通常用函数构造，它们通过一种严格的层次化方式相互调用。但解决某些问题时，有必要让函数调用自身。“递归函数”是一种能调用自身的函数——要么直接调用，要么间接调用。“递归”是高级计算机课程的一个重要主题。本节以及下一节要展示一些简单的递归例子。

我们先讨论递归的概念，然后再演示几个递归函数。用递归来解决问题的方式具有一些通用的元素。调用递归函数的目的是解决一个问题。函数实际只知道如何解决最简单的情况——或称“基本条件”。如果不是在基本条件下调用函数，函数会将问题分解成两个概念性的部分——一部分是函数知道怎样解决的（基本条件），另一部分是函数不知道的。为使递归可行，后者必须表面上“类似于”最初的问题，但稍简单一些，或稍小一些。由于这个新问题和原始问题类似，所以函数调用自己的一个新拷贝，对较小的问题进行处理——这称为“递归调用”（Recursive Call），也称为“递归步骤”（Recursion Step）。递归调用通常要使用关键字 return，因为结果要同函数已知怎样解决的那一部分问题合并，构成一个最终结果，再传回原调用者。

进行递归调用时，函数原始调用仍处于开放状态（也就是说，尚未结束执行）。递归调用可能造成

更多的递归调用，因为函数会将每个新的子问题都分解为两个概念性的部分。要想终止递归，不断变小的问题必须会聚成一个基本条件。此时，函数能识别出基本条件，并将结果返回前一个函数拷贝，并连续进行一系列返回，直到原始函数调用将最终结果返回调用者。同以前介绍的各种传统问题解决技术相比，这个过程听起来似乎有点儿复杂。为解释这些概念的实际运用，下面来写一个递归程序，执行一种流行的数学计算。

对非负整数 n 来说，它的阶乘写作 $n!$ （念作“ n 的阶乘”），公式如下：

$$n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 1$$

$1!$ 等于 1， $0!$ 等于 1。例如， $5!$ 等于 $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ ，结果为 120。

对于大于或等于 0 的整数 `number`，它的阶乘可通过 `for` 循环重复（非递归方式）来计算：

```
factorial = 1
for counter in range (1, number +1):
    factorial *= _counter
```

观察以下关系，可总结出阶乘的一个递归定义：

$$n! = n \cdot (n-1)!$$

例如， $5!$ 显然等于 $5 \cdot 4!$ ，如下所示：

$$\begin{aligned} 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ 5! &= 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \\ 5! &= 5 \cdot (4!) \end{aligned}$$

$5!$ 的计算过程如图 4.16 所示。图 4.16 中，(a) 展示了递归调用过程，直到 $1!$ 的结果为 1，此时会终止递归；(b) 显示了每次递归调用向其调用者返回的值，直到计算并返回终值。

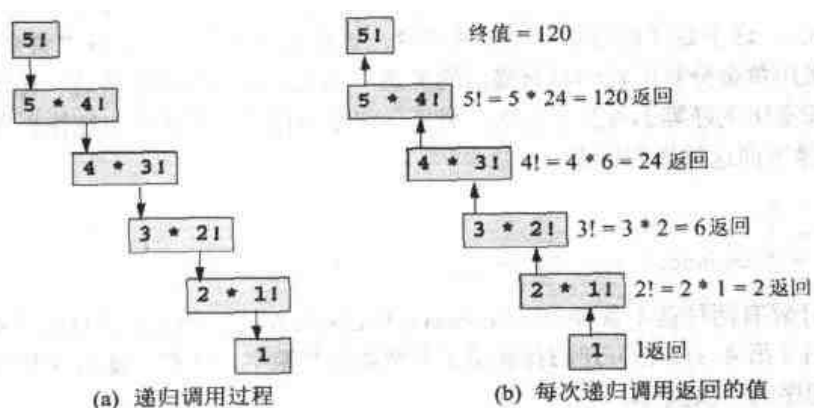


图 4.16 $5!$ 递归计算

图 4.17 利用递归来计算并打印从 0~10 的整数的阶乘。递归函数 `factorial`（第 5~10 行）首先检测终止条件是否为 `true`（第 7 行）。如果 `number` 小于或等于 1（基本条件），`factorial` 返回 1，不需要继续递归，函数将终止；如果 `number` 大于 1，第 10 行把问题表示为 `number` 与 `number - 1` 的阶乘的乘积。注意，`factorial (number - 1)` 是原始计算 `factorial (number)` 的一个稍简单的版本。

```
1 # Fig. 4.17: fig04_17.py
2 # Recursive factorial function.
3
4 # Recursive definition of function factorial
5 def factorial (number):
6
7     if number <= 1: # base case
```

```

8     return 1
9     else:
10        return number * factorial( number - 1 ) # recursive call
11
12 for i in range( 11 ):
13     print "%2d! = %d" % ( i, factorial( i ) )

```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

```

图 4.17 用于计算阶乘的递归函数

常见编程错误 4.7 如遗漏基本条件，或错误编写递归调用，就会造成最终无法会汇聚于基本条件，从而导致无穷递归，最终耗尽内存。这类似于重复（非递归）方案中的无限循环问题。

4.11 递归示例：斐波拉契序列

斐波拉契（Fibonacci）序列如下：

0, 1, 1, 2, 3, 5, 8, 13, 21……

序列肯定从 0 和 1 开始，后续每个斐波拉契数字都是前两个数字的和。

这种序列经常见于自然界，尤其是它描述了一种“螺旋分割”形式。相邻斐波拉契数字的比值逐渐接近一个固定值 1.618…。这个数字在自然界中经常出现，被称为“黄金比”或者“黄金分割比”。人类很早的时候就发现，采用黄金分割的物体具有强烈的美感。因此，建筑师通常按这一比例来设计窗户、房间等等，使它们的长宽比正好等于黄金分割比。另外，邮政明信片的长宽比也往往采用这个比率。

斐波拉契序列可像下面这样递归定义：

```

fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( n ) = fibonacci( n - 1 ) + fibonacci ( n - 2 )

```

注意，斐波拉契计算有两种基本条件——`fibonacci(0)`定义成 0，而 `fibonacci(1)`定义成 1。图 4.18 的程序使用函数 `fibonacci`（第 4~11 行）递归计算第 i 个斐波拉契数字。注意，斐波拉契数字增长很快。每个输出框都显示了程序的一次执行。

```

1 # Fig. 4.18: fig04_18.py
2 # Recursive fibonacci function.
3
4 def fibonacci( n ):
5
6     if n == 0 or n == 1: # base case
7         return n
8     else:
9
10        # two recursive calls
11        return fibonacci( n - 1 ) + fibonacci( n - 2 )
12
13 number = int( raw_input( "Enter an integer: " ) )
14
15 if number > 0:
16     result = fibonacci( number )
17     print "Fibonacci(%d) = %d" % ( number, result )

```

```

18 else:
19     print "Cannot find the fibonacci of a negative number"

```

```

Enter an integer: 0
Fibonacci(0) = 0

```

```

Enter an integer: 1
Fibonacci(1) = 1

```

```

Enter an integer: 2
Fibonacci(2) = 1

```

```

Enter an integer: 3
Fibonacci(3) = 2

```

```

Enter an integer: 4
Fibonacci(4) = 3

```

```

Enter an integer: 6
Fibonacci(6) = 8

```

```

Enter an integer: 10
Fibonacci(10) = 55

```

```

Enter an integer: 20
Fibonacci(20) = 6765

```

图 4.18 递归生成斐波拉契数字

对 fibonacci 的初始调用（第 16 行）不是递归调用，但从 fibonacci 主体对 fibonacci 的每次调用都是递归的。fibonacci 每次调用时，都会检测基本条件—— n 等于 0 或 1。如条件满足，fibonacci 返回 n （第 7 行）。有趣的是，如果 n 大于 1，递归调用会生成两个递归调用（第 11 行），与初始的 fibonacci 调用相比，分别都是一个更简单的问题。图 4.19 展示了计算 fibonacci(3) 的过程。

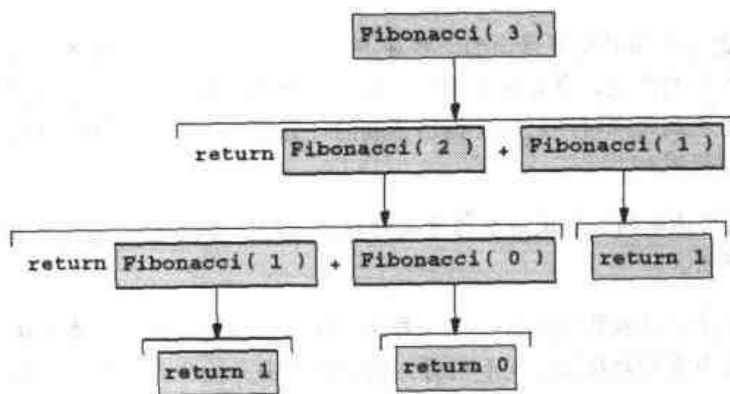


图 4.19 递归调用 fibonacci 函数

对于这个程序，必须注意一个重要问题。每次调用与某个基本条件（0 或 1）不符的 fibonacci 函数，都会另行生成两个对 fibonacci 的递归调用。这一系列递归调用很快就会失控。用图 4.18 的程序计算 20 的斐波拉契值，需调用 fibonacci 函数 21 891 次；如计算 30 的斐波拉契值，则需调用 2 692 537 次。

计算较大的斐波拉契值时，会注意到每个连续的斐波拉契数字都会导致计算时间和调用次数的显著增加。例如，31 的斐波拉契值需要 4 356 617 次调用，32 的斐波拉契值需要 7 049 155 次调用。如您所见，fibonacci 函数的调用次数会急剧增加——上例仅仅从 31 变成 32，便增加了 2 692 538 次调用。计算机科学家将这个问题称为“指数复杂性”，它甚至能让世界上最快的计算机甘拜下风！在“计算机算法”或“复杂性”等高级课程中，会详细讨论这个问题。

性能提示 4.2 一般不要编写会造成调用次数以指数级增加的“斐波拉契”式递归程序。

4.12 递归与重复

通过前面的学习，我们知道函数可采用递归或重复方式实现。本节要比较这两种方式，并探讨程序员在挑选具体方式时，主要应该考虑哪些因素。

递归与重复都建立在一个控制结构的基础上。重复使用的是重复结构，比如 for 和 while；递归则使用选择结构，比如 if 和 if/else。无论重复还是递归，实际都牵涉到重复性操作。区别在于，重复是显式使用一个重复结构，递归则是进行重复的函数调用。两者都要进行终止测试：重复会在循环继续条件为 false 时终止；递归则在识别出基本条件时终止。由计数器控制的重复和递归都是逐渐终止：重复会不断修改一个计数器，直到计数器的值使循环继续条件变成 false；递归则不断对原始问题进行简化，直到抵达基本条件。重复和递归都可能无休止地进行：如果循环继续检测永远不能变成 false，会发生无限循环；如果递归调用永远不能将问题简化成基本条件，会发生无穷递归。

递归有很多缺点。它采用重复调用机制，不断产生函数调用开销。这不仅浪费处理器处理时间，还会占用大量内存。每次递归调用都会导致创建另一个函数拷贝（事实只是函数的变量），这会占用大量内存。重复通常在函数内部发生，所以能忽略重复函数调用和额外内存分配的开销。既然如此，为什么还要选择递归？

软件工程知识 4.8 采用递归方式能解决的任何问题也可采用重复方式（非递归方式）解决。如果递归方式能够更自然地反映问题，并使程序易于理解和调试，通常应该首选递归方式。通常，只需几行代码即可实现一个递归方式。重复方式则相反，它需要大量代码来实现。选择递归的另一个原因是，重复方案也许不是很直观。

性能提示 4.3 避免在对性能要求较高的时候使用递归。递归调用既费时、又耗内存。

常见编程错误 4.8 让用于解决非递归算法的一个函数调用自身（无论直接还是间接）是逻辑错误。

这里不妨重新考虑一下本书反复强调的一些观点。良好的软件工程是重要的，高性能也是重要的。令人遗憾的是，它们不容易两全。良好软件工程是确保大型的、复杂的软件系统开发任务容易管理的一个关键。另一方面，考虑到将来可能对硬件提出更高的计算要求，这些系统的性能也不能太低。函数在此应该占据什么样的位置呢？

软件工程知识 4.9 采用清晰的、层次清楚的方式对程序进行“函数化”，有助于保证良好的软件工程，但性能上要付出一定代价。

性能提示 4.4 一个由多个函数构成的程序——与没有任何函数的一体式程序相比——会产生大量的函数调用，这些调用会占用大量处理器时间和内存。但另一方面，一体式程序的编程、测试、调试和维护都比较复杂。

因此，对程序进行函数化时要综合考虑。要根据实际情况，保证能同时兼顾良好的性能和软件工程。

4.13 默认参数

函数调用通常要传递参数值。定义函数时，程序员可将一个参数指定为“默认参数”，并可为其指定一个默认值。默认参数主要是为了提供方便。调用函数时，可指定较少的参数数量。在函数调用中省略默认参数，解释器会插入它的默认值，并在调用中传递参数。

在参数列表中，默认参数必须位于任何非默认参数的右边。调用有两个或更多默认参数的函数时，

如果省略的参数不在参数列表最右边，那就必须省略它之后的所有参数。

图 4.20 演示了默认参数在计算箱子容积时的应用。第 5 行的 `boxVolume` 函数定义将三个参数的默认值都设为 1。注意，只能在函数的 `def` 语句中定义默认值。

```

1 # Fig. 4.20: fig04_20.py
2 # Using default arguments.
3
4 # function definition with default arguments
5 def boxVolume( length = 1, width = 1, height = 1 ):
6     return length * width * height
7
8 print "The default box volume is:", boxVolume()
9 print "\nThe volume of a box with length 10,"
10 print "width 1 and height 1 is:", boxVolume( 10 )
11 print "\nThe volume of a box with length 10,"
12 print "width 5 and height 1 is:", boxVolume( 10, 5 )
13 print "\nThe volume of a box with length 10,"
14 print "width 5 and height 2 is:", boxVolume( 10, 5, 2 )

```

```

The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100

```

图 4.20 默认参数

`boxVolume` 的第一次调用（第 8 行）没有指定参数，因此三个参数都采用默认值。第二次调用（第 10 行）传递了一个 `length` 参数，所以 `width` 和 `height` 参数使用默认值。第三次调用（第 12 行）传递了 `length` 和 `width` 参数，所以 `height` 参数使用默认值。最后一次调用（第 14 行）传递了 `length`、`width` 和 `height` 参数，所以没有使用默认值。

良好编程习惯 4.6 使用默认参数可简化函数调用的编写。但有的程序员认为，显式指定所有参数会使程序更易读。

常见编程错误 4.9 默认参数必须全部靠右。省略非靠右的参数是语法错误。

4.14 关键字参数

程序员可要求函数接收一个或多个“关键字参数”。函数定义为每个关键字指派一个默认值。函数既可使用关键字的默认值，也可在函数调用中为关键字指派一个新值（格式为 `keyword = value`）。使用关键字参数时，参数在函数调用中的位置不必与它在函数定义中的位置一一对应。图 4.21 演示了如何在 Python 程序中使用关键字参数显示同一个请求的网站有关信息。

```

1 # Fig. 4.21: fig04_21.py
2 # Keyword arguments example.
3
4 def generateWebsite( name, url = "www.deitel.com",
5     Flash = "no", CGI = "yes" ):
6     print "Generating site requested by", name, "using url", url
7
8     if Flash == "yes":
9         print "Flash is enabled"
10
11     if CGI == "yes":
12         print "CGI scripts are enabled"

```



```

13     print # prints a new line
14
15     generateWebsite( "Deitel" )
16
17     generateWebsite( "Deitel", Flash = "yes",
18         url = "www.deitel.com/new" )
19
20     generateWebsite( CGI = "no", name = "Prentice Hall" )

```

```

Generating site requested by Deitel using url www.deitel.com
CGI scripts are enabled

Generating site requested by Deitel using url www.deitel.com/new
Flash is enabled
CGI scripts are enabled

Generating site requested by Prentice Hall using url www.deitel.com

```

图 4.21 关键字参数

`generateWebsite` 取得 4 个参数。关键字参数的名称为 `url`, `Flash` 和 `CGI`, 它们的默认值分别为 "www.deitel.com", "no" 和 "yes" (第 4~5 行)。函数判断是谁请求网站, 并在网站支持 `Flash` 或 `CGI` 的前提下显示一条消息 (第 6~13 行)。

第 15 行的函数调用仅向 `generateWebsite` 函数传递了一个参数值, 即参数 `name` 的值。其余参数将使用定义中给定的默认值。

第 17~18 行的函数调用向 `generateWebsite` 传递 3 个参数。其中, 变量 `name` 的值同样是 "Deitel"。此外, 还将值 "yes" 指派给关键字参数 `Flash`, 将 "www.deitel.com/new" 指派给关键字参数 `url`。这个函数调用证明了关键字参数的顺序比普通参数灵活得多。Python 解释器必须根据 "Deitel" 在函数调用中的位置才能将其与变量 `name` 匹配。相反, 对于传给 `url` 和 `Flash` 的值, 则根据它们的关键字参数名来匹配, 而非根据位置。在对 `generateWebsite` 的任何调用中, `name` 的值必须排在第一位 (只要不在参数列表中通过为 `name` 指派值以引用它)。第 20 行则演示了任何参数实际都可当作一个关键字来引用, 即使它没有默认值。

图 4.22 的交互式会话演示了混合使用非关键字和关键字参数时, 最容易犯的一些错误。函数调用 `test(number = "two", "Name")` 之所以出错, 是由于将非关键字参数放到了关键字参数之后。函数调用 `test(number = "three")` 也是不正确的, 因为 `test` 函数期待的是一个非关键字参数。

常见编程错误 4.10 在函数调用中错误放置或遗漏非关键字参数的值是错误的。

```

Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> def test( name, number1 = "one", number2 = "two" ):
...     pass
...
>>> test( number1 = "two", "Name" )
SyntaxError: non-keyword arg after keyword arg
>>> test( number1 = "three" )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: test() takes at least 1 non-keyword argument (0 given)

```

图 4.22 关键字参数的错误

第5章 列表、元组和字典

学习目标

- 理解 Python 序列
- 介绍列表、元组和字典数据类型
- 理解如何创建、初始化和引用列表、元组和字典的单独元素
- 理解如何通过列表来排序和搜索值序列
- 学会将列表传给函数
- 介绍列表和字典方法
- 会创建和处理多下标列表和元组

5.1 概述

本章介绍 Python 通过“数据结构”来实现的数据处理能力。数据结构用于容纳和组织信息（数据）。数据结构的类型很多，每种类型都适合完成特定任务。其中，“序列”在其他语言中通常称为“数组”，是用于存储相关数据项的数据结构；Python 支持 3 种基本序列数据类型：字符串（string）、列表（list）和元组（tuple）。“映射”在其他语言中通常称为关联数组或“哈希”，是用于存储“键-值对”对的数据结构。Python 支持一种映射数据类型：字典（dictionary）。本章要通过几个例子讨论 Python 的序列和映射类型。第 22 章要介绍一些高级数据结构（链表、序列、堆栈和树），它们对 Python 的基本数据类型进行了扩展。

5.2 序列

序列是一系列连续值，它们通常是相关的。前几个程序中已经出现过序列：Python 字符串是序列，range 函数返回的值也是（range 是 Python 内建函数，可返回一系列整数）。本节要详细讨论序列，并解释如何引用序列中的特定元素（或位置）。

图 5.1 展示了序列 c，它包含 12 个整数元素。引用元素时，可先写下序列名，再在方括号中写下元素的位置编号。序列的第 1 个元素称为“第零个元素”。所以在序列 c 中，第 1 个元素是 c[0]，第 2 个是 c[1]，第 6 个是 c[5]。通常，序列 c 的第 i 个元素是 c[i-1]。

序列也可从尾部访问。最后一个元素是 c[-1]，倒数第二个是 c[-2]，倒数第 i 个元素是 c[-i]。序列的命名规范同变量是一样的。

位置编号更正式的称呼是“下标”或“索引”，它必须是整数或整数表达式。如果程序将整数表达式用作下标，Python 就会对表达式进行求值，以确定索引。例如，假定变量 a 等于 5，变量 b 等于 6，以下语句：

```
print c[ a + b ]
```

会打印 c[11] 的值。通过循环来遍历一个序列时，将整数表达式用作下标特别有用。

Python 列出的字典是“可变”的——可对其进行修改。例如，假定图 5.1 中，序列 c 可变，以下语句：

```
c[ 11 ] = 0
```


会为元素 11 指派新值 0，替换原来的值 78，从而修改该元素的值。

序列名称 (c)



c[0]	-45	c[-12]
c[1]	6	c[-11]
c[2]	0	c[-10]
c[3]	72	c[-9]
c[4]	1543	c[-8]
c[5]	-89	c[-7]
c[6]	0	c[-6]
c[7]	62	c[-5]
c[8]	-3	c[-4]
c[9]	1	c[-3]
c[10]	6453	c[-2]
c[11]	78	c[-1]

元素在序列 c 中的位置编号

图 5.1 含有元素和索引的序列

另一方面，某些类型的序列是“不可变”的——不可对其进行修改，或者说不可更改元素值。Python 字符串和元组都属于不可变序列。例如，假定序列 c 是不可变的，以下语句：

```
c[ 11 ] = 0
```

就是非法的。下面详细研究一下序列 c。该序列的名称是 c，序列长度可通过函数调用 len(c) 来确定。经常都需要知道序列长度，因为假如引用序列外部的一个元素，会导致“越界”（Out-of-Range）错误。本章讨论的大多数错误都可作为异常加以捕捉。第 12 章会详细讨论异常问题。

序列 c 包含 12 个元素，即 c[0]，c[1]，…，c[11]。元素范围也可引用为 c[-12]，c[-11]，…，c[-1]。在本例，c[0] 包含值 -45，c[1] 包含值 6，c[-9] 包含值 72，而 c[-2] 包含值 6453。为了对前三个元素的值求和，并将结果指派给变量 sum，可使用以下语句：

```
sum = c[ 0 ] + c[ 1 ] + c[ 2 ]
```

为了将第 7 个元素的值除以 2，并将结果指派给变量 x，可使用以下语句：

```
x = c[ 6 ] / 2
```

常见编程错误 5.1 注意“序列第 7 个元素”和“序列元素 7”的差异。序列下标自 0 开始，所以“序列第 7 个元素”的下标是 6。相反，“序列元素 7”的下标就是 7，即 c[7]，而且实际是序列的第 8 个元素。如果混淆这两种说法，就会导致“相差 1”错误。

测试和调试提示 5.1 在不接受负下标的其他程序语言中，如果不慎计算出负下标，会导致运行时错误。在 Python 中，不慎出现的负下标会导致非严重逻辑错误，程序继续运行，只是产生无效结果。

用于封闭序列下标的方括号实际是 Python 运算符。图 5.2 总结了前文已经介绍过的 Python 运算符的优先级和顺序关联性。各运算符根据优先级从上到下降序排列。

运算符	顺序关联性	类型
()	从左到右	圆括号
[]	从左到右	下标
.	从左到右	成员访问
**	从右到左	求幂
* / // %	从左到右	乘
+ -	从左到右	加
< <= > >=	从左到右	关系
== != <>	从左到右	相等
and	从左到右	逻辑 AND
or	从左到右	逻辑 OR
not	从右到左	逻辑 NOT

图 5.2 运算符的优先级和顺序关联性

5.3 创建序列

不同的 Python 序列（字符串、列表和元组）需要不同的语法。以前介绍了如何将字符串文本放到引号内，以创建 Python 字符串。要创建空字符串，请使用以下语句：

```
aString = ""
```

注意，创建字符串时，还可使用单引号（'）或三引号（"""或'''）。创建空列表的语句是：

```
aList = []
```

要创建包含值序列的一个列表，在方括号内用逗号分隔不同的值：

```
aList = [ 1, 2, 3 ]
```

要创建空元组，使用以下语句：

```
aTuple = ()
```

要创建包含值序列的一个元组，只需用逗号分隔不同的值：

```
aTuple = 1, 2, 3
```

创建元组的过程也称为“元组打包”（packing a tuple）。另外，可用可选的圆括号封闭用逗号来分隔的元组值列表，以创建一个元组。但要注意，创建元组的是逗号，而非圆括号。

```
aTuple = ( 1, 2, 3 )
```

创建单元素元组时（称为 Singleton），请使用以下语句：

```
aSingleton = 1,
```

注意，值后要插入一个逗号，逗号用于将变量 aSingleton 标识为元组。遗失逗号，aSingleton 会变成包含整数值 1 的简单变量。

5.4 使用列表和元组

列表和元组都包含值序列。例如，列表或元组可包含从 1~5 的整数序列，如下所示：

```
aList = [ 1, 2, 3, 4, 5 ]
aTuple = ( 1, 2, 3, 4, 5 )
```

但实际上，Python 程序员经常对这两种数据类型进行区分，用它们表示不同种类的序列，具体取决于程序的实际情况。下一节要讨论列表和元组适用于哪些情况。

5.4.1 使用列表

尽管不限制列表只能包含同种数据类型（即要求所有值都具有相同数据类型），但 Python 程序员通常都用列表存储同种值的序列。例如，用一个列表存储整数序列，表示测验成绩；用另一个字符串序列表示员工姓名。通常，程序用列表来存储同种值，以便遍历这些值，并对每个值都执行相同的操作。列表长度一般是事先未确定的，并可在程序执行期间发生改变。图 5.3 的程序演示了如何创建一个列表，并在其中添加和获取值。

```
1 # Fig. 5.3: fig05_03.py
2 # Creating, accessing and changing a list.
3
4 aList = [] # create empty list
5
6 # add values to list
7 for number in range( 1, 11 ):
8     aList += [ number ]
9
10 print "The value of aList is:", aList
11
12 # access list values by iteration
13 print "\nAccessing values by iteration:"
14
15 for item in aList:
16     print item,
17
18 print
19
20 # access list values by index
21 print "\nAccessing values by index:"
22 print "Subscript Value"
23
24 for i in range( len( aList ) ):
25     print "%9d %7d" % ( i, aList[ i ] )
26
27 # modify list
28 print "\nModifying a list value..."
29 print "Value of aList before modification:", aList
30 aList[ 0 ] = -100
31 aList[ -3 ] = 19
32 print "Value of aList after modification:", aList
```

```
The value of aList is: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
Accessing values by iteration:
1 2 3 4 5 6 7 8 9 10
```

```
Accessing values by index:
```

Subscript	Value
0	1
1	2
2	3
3	4
4	5

```

5      6
6      7
7      8
8      9
9     10

Modifying a list value...
Value of alist before modification: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Value of alist after modification: [-100, 2, 3, 4, 5, 6, 7, 19, 9, 10]

```

图 5.3 同种值的列表

第4行创建空列表 `alist`。第7~8行用 `for` 循环在 `alist` 中插入1~10的值，使用的是+=自增赋值语句。+=左边的值如果是一个序列，右边的值也必须是一个序列。所以，第8行用方括号封闭了要添加到列表的值。第10行打印变量 `alist`。Python 将列表显示成用逗号分隔的值序列。变量 `alist` 代表一个典型 Python 列表——它是包含同种数据的一个序列。

第13~18行演示了访问列表元素的最常用的方式。`for` 结构实际要遍历一个序列：

```
for item in alist:
```

`for` 结构（第15~16行）从序列的第一个元素开始，将它的值指派给控制变量 `item`，并执行 `for` 循环主体（即打印控制变量的值）。之后，循环继续处理下一个元素，并执行相同的操作。所以，第15~16行会打印 `alist` 的每个元素。

列表元素也可通过其索引来访问。第21~25行用这种方式访问 `alist` 中的每个元素。第24行的函数调用：

```
range( len( alist ) )
```

会返回一个序列，其中包含值0, ..., `len(alist) - 1`。这个序列包含了 `alist` 所有可能的元素位置。`for` 循环遍历这个序列，并针对每个元素位置，打印出位置以及存储在那个位置的值。

第30~31行修改列表的一些元素。为修改特定元素值，我们为元素分配一个新值。第30行将列表第一个元素的值从0变为-100；第31行则将列表倒数第三个元素从8变为19。

如果试图访问 `alist` 中不存在的索引（比如索引13），程序会退出，Python 将显示一条越界错误消息（`List index out of range`）。图5.4的交互式会话展示了访问越界列表元素的结果。

```

Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> alist = [ 1 ]
>>> print alist[ 13 ]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range

```

图 5.4 越界错误

常见编程错误 5.2 引用序列之外的元素是错误的。

测试和调试提示 5.2 遍历序列时，正的序列下标应小于序列元素总数（换言之，下标不能大于序列长度）。另一方面，负的序列下标应等于或大于序列元素总数的负数。循环终止条件应能避免访问超出这个范围的元素。

通常，程序本身不关心列表长度，只需遍历列表，并执行其中每个元素执行操作。图5.5演示了如此使用列表的一个例子——它根据一系列数据创建柱形图。

```

1 # Fig. 5.5: fig05_05.py
2 # Creating a histogram from a list of values.
3

```

```

4 values = [] # a list of values
5
6 # input 10 values from user
7 print "Enter 10 integers:"
8
9 for i in range( 10 ):
10     newValue = int( raw_input( "Enter integer %d: " % ( i + 1 ) ) )
11     values += [ newValue ]
12
13 # create histogram
14 print "\nCreating a histogram from values:"
15 print "%s %10s %10s" % ( "Element", "Value", "Histogram" )
16
17 for i in range( len( values ) ):
18     print "%7d %10d %s" % ( i, values[ i ], "*" * values[ i ] )

```

```

Enter 10 integers:
Enter integer 1: 19
Enter integer 2: 3
Enter integer 3: 15
Enter integer 4: 7
Enter integer 5: 11
Enter integer 6: 9
Enter integer 7: 13
Enter integer 8: 5
Enter integer 9: 17
Enter integer 10: 1

Creating a histogram from values:
Element      Value Histogram
0            19 *****
1             3 ***
2            15 *****
3             7 *****
4            11 *****
5             9 *****
6            13 *****
7             5 *****
8            17 *****
9             1 *

```

图 5.5 根据值列表创建柱形图

第 4 行创建空列表 `values`。第 7~11 行从用户处输入 10 个整数，并将它们存储到列表。第 14~18 行创建柱形图。针对每个列表元素，都打印它的索引和值，再打印一个星号（*）字符串；星号的数量与值相等。以下表达式：

```
"*" * values[ i ]
```

使用乘法运算符（*）来创建一个字符串，其中包含了数量由 `values[i]` 指定的星号。

5.4.2 使用元组

列表存储的通常是同种数据的序列，相反，元组通常存储异种数据的序列——但这只是一种习惯，而非规则。元组的每个数据项都提供了元组所表示的总体信息中的一部分。假定一个元组表示某个班的一名学生，那么元组中可包含学生的姓名（表示成字符串）和年龄（表示成整数）。另外，也可用元组表示一天中的时间。时间由 3 部分构成：小时、分钟和秒。尽管所有值都可用整数表示，但每个整数都有自己的含义，要想完整获取时间信息，只需将 3 个值都提取出来。元组长度（即数据项的数量）是事先确定的，不可在程序执行期间更改。

按照约定，元组中每个数据项都代表总体数据的一个特定部分。所以，程序通常不遍历元组，而是只访问完成当前任务所需的元组的一部分。图 5.6 展示了如何创建和访问元组。

```

1 # Fig. 5.6: fig05_06.py
2 # Creating and accessing tuples.
3
4 # retrieve hour, minute and second from user
5 hour = int( raw_input( "Enter hour: " ) )
6 minute = int( raw_input( "Enter minute: " ) )
7 second = int( raw_input( "Enter second: " ) )
8
9 currentTime = hour, minute, second # create tuple
10
11 print "The value of currentTime is:", currentTime
12
13 # access tuple
14 print "The number of seconds since midnight is", \
15       ( currentTime[ 0 ] * 3600 + currentTime[ 1 ] * 60 +
16         currentTime[ 2 ] )

```

```

Enter hour: 9
Enter minute: 16
Enter second: 1
The value of currentTime is: (9, 16, 1)
The number of seconds since midnight is 33361

```

图 5.6 创建和访问元组

第5~7行要求用户输入3个整数，分别表示小时、分钟和秒。第9行创建名为 `currentTime` 的元组，在其中存储用户输入的值。第4~16行打印自午夜经历的秒数。对于元组中的每个值，我们都采取了不同的操作（具体是让每个值都乘以不同的因子）。因此，程序根据索引来访问每个值。

元组是不可变的。Python 提供了错误处理机制，一旦发现可能会修改元组，就会通知用户。例如，假定程序试图更改 `currentTime` 的第一个元素，令其包含值 0，如下所示：

```
currentTime[ 0 ] = 0
```

程序就会退出，Python 将报告以下运行时错误

```

Traceback (most recent call last):
  File "fig05_06.py", line 18, in ?
    currentTime[ 0 ] = 0
TypeError: object doesn't support item assignment

```

指出程序试图非法更改不可变元组的值。

注意，5.4.1 节和 5.4.2 节所介绍的列表和元组的用法并不是一种硬性的规则，只是 Python 程序员遵守的一种约定。Python 并不限制存储在列表和元组中的数据类型（也就是说，它们能任意包含同种或异种数据）。列表和元组的主要区别在于，列表是可变的，元组则是不可变的。

5.4.3 序列解包

记住，如果用以下语句：

```
aTuple = 1, 2, 3
```

或者用以下语句：

```
aTuple = ( 1, 2, 3 )
```

来创建元组，就称为“元组打包”，因为值被“打包”到元组中。元组和其他序列还可以“解包”——将序列中存储的值指派给各个标识符。通过“解包”而在单个语句中为多个变量指派值，是一种简化编程的有效手段。图 5.7 的程序演示了对字符串、列表和元组进行解包的结果。

```

1 # Fig. 5.7: fig05_07.py
2 # Unpacking sequences.
3
4 # create sequences
5 aString = "abc"
6 aList = [ 1, 2, 3 ]
7 aTuple = "a", "A", 1
8
9 # unpack sequences to variables
10 print "Unpacking string..."
11 first, second, third = aString
12 print "String values:", first, second, third
13
14 print "\nUnpacking list..."
15 first, second, third = aList
16 print "List values:", first, second, third
17
18 print "\nUnpacking tuple..."
19 first, second, third = aTuple
20 print "Tuple values:", first, second, third
21
22 # swapping two values
23 x = 3
24 y = 4
25
26 print "\nBefore swapping: x = %d, y = %d" % ( x, y )
27 x, y = y, x      # swap variables
28 print "After swapping: x = %d, y = %d" % ( x, y )

```

```

Unpacking string...
String values: a b c

Unpacking list...
List values: 1 2 3

Unpacking tuple...
Tuple values: a A 1

Before swapping: x = 3, y = 4
After swapping: x = 4, y = 3

```

图 5.7 字符串、列表和元组解包

第 5~7 行创建一个字符串、一个列表和一个元组，各自都包含 3 个元素。序列是通过赋值语句来解包的。第 11 行的赋值语句对变量 `aString` 的元素进行解包，将每个元素都指派给一个变量。第一个元素指派给变量 `first`，第二个指派给 `second`，第三个指派给 `third`。第 12 行打印这些变量，证明字符串已成功解包。第 14~20 行对变量 `aList` 和 `aTuple` 中的元素执行类似的操作。解包一个序列时，运算符 `=` 左侧的变量名数量应等于右侧那个序列所包含的元素个数；否则会产生运行时错误。注意，解包序列时，可在运算符 `=` 左侧选择添加圆括号或方括号，因为通常不存在优先顺序问题。

第 22~28 行演示了序列打包和解包的一种应用——交换两个变量的值。第 23~24 行创建两个变量 `x` 和 `y`，分别指派值 3 和值 4。第 27 行：

```
x, y = y, x
```

对指派给每个变量的值进行交换。Python 在交换值时，首先将右边的部分打包成一个元组，即 `(4, 3)`。然后，将该元组解包成变量 `x` 和 `y`。所以，以前指派给变量 `x` 的值会被指派给变量 `y`，指派给变量 `y` 的值会被指派给变量 `x`。

5.4.4 序列分片

前面讨论了如何创建序列，如何用 `[]` 运算符访问一个元素，以及如何用 `for` 语句遍历所有元素。有

时，程序需要访问一系列连续的值（比如在用于存储姓名的一个字符串中，只访问构成姓氏的那一部分字符）。针对这些情况，Python 允许使用序列“分片”（slice）。

图 5.8 演示了 Python 的序列分片能力。程序创建 3 个序列：一个字符串、一个元组和一个列表。程序提示用户输入起始和结束索引，为每个序列创建指定的分片，然后在屏幕上打印分片结果。

```

1 # Fig. 5.8: fig05_08.py
2 # Slicing sequences.
3
4 # create sequences
5 sliceString = "abcdefghij"
6 sliceTuple = ( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 )
7 sliceList = [ "I", "II", "III", "IV", "V",
8               "VI", "VII", "VIII", "IX", "X" ]
9
10 # print strings
11 print "sliceString: ", sliceString
12 print "sliceTuple: ", sliceTuple
13 print "sliceList: ", sliceList
14 print
15
16 # get slices
17 start = int( raw_input( "Enter start: " ) )
18 end = int( raw_input( "Enter end: " ) )
19
20 # print slices
21 print "\nsliceString[", start, ":", end, "] = ", \
22       sliceString[ start:end ]
23
24 print "sliceTuple[", start, ":", end, "] = ", \
25       sliceTuple[ start:end ]
26
27 print "sliceList[", start, ":", end, "] = ", \
28       sliceList[ start:end ]

```

```

sliceString: abcdefghij
sliceTuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
sliceList: ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII',
            'IX', 'X']

```

```

Enter start: 3
Enter end: 3

```

```

sliceString[ 3 : 3 ] =
sliceTuple[ 3 : 3 ] = ()
sliceList[ 3 : 3 ] = []

```

```

sliceString: abcdefghij
sliceTuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
sliceList: ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII',
            'IX', 'X']

```

```

Enter start: -4
Enter end: -1

```

```

sliceString[ -4 : -1 ] = ghi
sliceTuple[ -4 : -1 ] = (7, 8, 9)
sliceList[ -4 : -1 ] = ['VII', 'VIII', 'IX']

```

```

sliceString: abcdefghij
sliceTuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
sliceList: ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII',
            'IX', 'X']

```

```

Enter start: 0
Enter end: 10

```

```

sliceString[ 0 : 10 ] = abcdefghij
sliceTuple[ 0 : 10 ] = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

```



```
sliceList[ 0 : 10 ] = ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII',
'VIII', 'IX', 'X']
```

图 5.8 序列分片

第 15~18 行创建 3 个序列，并请求用户指定分片的开始与结束索引。第 21~28 行打印每个序列的指定分片。分片实际上就是一个新序列，它基于现有序列创建。第 22 行的表达式：

```
sliceString[ start:end ]
```

创建了变量 sliceString 的一个新序列，其中包含存储在索引位置 sliceString[start], ..., sliceString[end - 1] 的值。通常情况下，要想从序列中获取从第 *i* 个元素到第 *j* 个元素的一个分片（其中包含第 *i* 个和第 *j* 个元素），要使用以下表达式：

```
sequence[ i:j + 1 ]
```

图 5.8 展示了程序的 4 次示范输出。最后一次输出创建从索引 0~10 的一个分片（即整个序列）。记住，所有序列的第一个元素的索引都是 0。从这个分片创建的序列等价于用以下表达式创建的序列：

```
sequence[ : ]
```

上述表达式会新建一个序列，它是原始序列的完整拷贝。上述表达式还等价于：

```
sequence[ 0 : len( sequence ) ]
sequence[ : len( sequence ) ]
sequence[ 0 : ]
```

利用序列分片语法，可方便地选择现有序列的一部分。程序将列表传给函数时，可用序列分片技术创建列表的一个拷贝。这方面的问题将在 5.7 节和 5.8 节讲解。

注意，负的分片不能直接访问列表的最后一个元素（sliceString[-4 : -1] = ghi），因为分片应用于元素之间的位置。使用负的分片，元素之间的最后一个位置是索引-2 和-1 的元素之间的位置。

5.5 字典

除了列表和元组，Python 还支持另一种功能强大的数据类型，即“字典”（Dictionary）。字典在其他语言中一般称为哈希或关联数组，它是一种“映射”结构，由“键-值对”构成。可将字典想象成一个集合，该集合由无序的值构成。每个值都通过它相应的键来引用。比如，在一个存储有电话号码的字典中，可根据一个人的姓名来引用电话号码。以下语句：

```
emptyDictionary = {}
```

创建了一个空字典。注意字典用花括号（{}）来注明。要想为字典初始化键-值对，需要使用以下语句：

```
dictionary = { 1 : "one", 2 : "two" }
```

每个键-值对都采取以下形式：

key : *value*

逗号用于分隔每个键-值对。字典的“键”必须是不可变的值，比如字符串、数字或元组；字典的“值”则可为任意 Python 数据类型。

常见编程错误 5.3 将列表或字典用作一个字典的“键”是语法错误。

图 5.9 演示了如何创建、初始化、访问和操纵简单的字典。第 5~6 行创建并打印一个空字典。第 9

行创建字典 `grades`，并初始化它，在其中包含 4 个键-值对。键是包含学生姓名的字符串，整数值则代表学生的成绩。第 10 行打印指派给变量 `grades` 的值。注意在显示 `grades` 时，顺序与定义时的顺序不同。这是因为字典属于无序的键-值对集合。另外注意在输出中，字典的键出现在单引号内。这是因为 Python 用单引号显示字符串。

```

1 # Fig. 5.9: fig05_09.py
2 # Creating, accessing and modifying a dictionary.
3
4 # create and print an empty dictionary
5 emptyDictionary = {}
6 print "The value of emptyDictionary is:", emptyDictionary
7
8 # create and print a dictionary with initial values
9 grades = { "John": 87, "Steve": 76, "Laura": 92, "Edwin": 89 }
10 print "\nAll grades:", grades
11
12 # access and modify an existing dictionary
13 print "\nSteve's current grade:", grades[ "Steve" ]
14 grades[ "Steve" ] = 90
15 print "Steve's new grade:", grades[ "Steve" ]
16
17 # add to an existing dictionary
18 grades[ "Michael" ] = 93
19 print "\nDictionary grades after modification:"
20 print grades
21
22 # delete entry from dictionary
23 del grades[ "John" ]
24 print "\nDictionary grades after deletion:"
25 print grades

```

```

The value of emptyDictionary is: {}

All grades: {'Edwin': 89, 'John': 87, 'Steve': 76, 'Laura': 92}

Steve's current grade: 76
Steve's new grade: 90

Dictionary grades after modification:
{'Edwin': 89, 'Michael': 93, 'John': 87, 'Steve': 90, 'Laura': 92}

Dictionary grades after deletion:
{'Edwin': 89, 'Michael': 93, 'Steve': 90, 'Laura': 92}

```

图 5.9 创建、访问和修改字典

第 13 行访问特定的字典值，这要用到运算符 `[]`。字典值要用以下形式的表达式访问：

```
dictionaryName[ key ]
```

在第 13 行，`dictionaryName` 是 `grades`，而 `key` 是字符串 "Steve"。该表达式的求值结果是存储在字典键 "Steve" 处的值，即 76。第 14 行将新值 90 指派给键 "Steve"。修改字典值所用的语法类似于修改列表的语法。第 15 行打印字典值改变后的结果。第 18 行在字典里插入一个新的键-值对。尽管所用语法同修改现有字典值的语法相似，但它能插入新的键-值对，因为 Michael 是一个新键。以下语句：

```
dictionaryName[ key ] = value
```

能修改对应于 `key` 的 `value`，前提是字典必须包含那个 `key`。否则，上述语句会在字典里插入新的键-值对。

软件工程知识 5.1 在字典里添加键-值对时，键名拼写错误会导致难以发现的逻辑错误。

第 19~20 行打印在字典里添加新的键-值对的结果。打印顺序完全是任意的（记住，字典是键-值对的一个无序集合）。表达式 `dictionaryName[key]` 可能导致不易察觉的编程错误。如果该表达式出现在赋

值语句的左侧，但字典里不包含那个键，赋值语句就会在字典中插入该键 - 值对。然而，如果表达式出现在赋值语句（或试图访问存储于指定键处的值的任何语句）右侧，会导致程序终止，并显示一条错误消息，因为程序试图访问一个不存在的键。

常见编程错误 5.4 试图访问不存在的字典键会引发 `KeyError` 异常，这是一种运行时错误。

第 23 行从字典中删除一项，以下语句：

```
del dictionaryName[ key ]
```

可从字典删除指定键及其值。如指定的键不存在，上述语句会导致程序终止，并显示一条错误消息。同样，这是因为程序试图访问不存在的键。这种运行时错误可通过异常处理进行捕捉，详情参见第 12 章。

字典是一种功能强大的数据类型，能帮助程序员完成复杂任务。许多 Python 模块都提供了类似于字典的数据结构，以简化对较复杂的数据类型的访问及操纵。下一节将继续探讨字典的功能。

5.6 列表和字典方法

通过前面的学习，我们知道序列和字典使程序员能完成一些高级数据处理，比如存储和获取数据等等。现在要介绍一种新的编程概念，即“方法”，它对数据处理功能进行了扩展。

如第 2 章所述，所有 Python 数据类型都至少包含 3 个属性：一个值、一个类型以及一个位置。有的 Python 数据类型（比如字符串、列表和字典）还包含方法。方法其实就是函数，负责执行对象的特定行为（任务）。本节讨论了列表和字典方法；字符串的方法将在第 13 章讨论。

列表方法实现了几种行为，比如将值附加到列表末尾，或者确定特定元素在列表中的索引等等。图 5.10 的程序使用一个列表方法将数据项附加到列表末尾。程序要求用户输入莎士比亚戏剧名称，并将名称附加到列表。第 4 行创建空列表 `playList`，用它存储用户输入的戏剧名。`for` 结构（第 8~10 行）使用列表方法 `append` 将数据项附加到变量 `playList` 的末尾。为调用列表方法，需要指定列表名，添加一个点访问运算符（`.`），再添加方法调用（即方法名和必要的参数）。第 14~15 行定义另一个 `for` 循环来打印用户输入的戏剧名称。注意，第 15 行用格式化字符（`-`）使名称左对齐。

```
1 # Fig. 5.10: fig05_10.py
2 # Appending items to a list.
3
4 playList = []      # list of favorite plays
5
6 print "Enter your 5 favorite Shakespearean plays.\n"
7
8 for i in range( 5 ):
9     playName = raw_input( "Play %d: " % ( i + 1 ) )
10    playList.append( playName )
11
12 print "\nSubscript      Value"
13
14 for i in range( len( playList ) ):
15     print "%9d      %-25s" % ( i + 1, playList[ i ] )
```

```
Enter your 5 favorite Shakespearean plays.
```

```
Play 1: Richard III
Play 2: Henry V
Play 3: Twelfth Night
Play 4: Hamlet
Play 5: King Lear
```

```
Subscript      Value
      1      Richard III
      2      Henry V
      3      Twelfth Night
```

```

4 Hamlet
5 King Lear

```

图 5.10 将数据项附加到列表

图 5.10 演示了如何通过数据类型的方法来执行有用的数据处理任务。图 5.11 则用另一个列表方法执行更典型的数据处理任务——统计特定的值在列表中出现的次数。第 4~7 行创建列表 `responses`，其中包含 1~10 的几个值。第 11~12 行包含一个 `for` 循环，它调用列表方法 `count` 来返回元素在列表中出现的次数。方法 `count` 需要取得一个参数，它可为任意数据类型。如列表的任何元素都不具有指定的值，`count` 会返回 0。第 11~12 行打印每个值在列表中出现的频率。

```

1 # Fig. 5.11: fig05_11.py
2 # Student poll program.
3
4 responses = [ 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
5              1, 6, 3, 8, 6, 10, 3, 8, 2, 7,
6              6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
7              5, 6, 7, 5, 6, 4, 8, 6, 8, 10 ]
8
9 print "Rating      Frequency"
10
11 for i in range( 1, 11 ):
12     print "%6d %13d" % ( i, responses.count( i ) )

```

```

Rating      Frequency
1           2
2           2
3           2
4           2
5           5
6          11
7           5
8           7
9           1
10          3

```

图 5.11 列表方法 `count`

列表还提供了另外几个有用的方法，图 5.12 对其进行了总结。后文新建的程序将经常调用列表方法。

方法	说明
<code>append(item)</code>	在列表末尾插入 <code>item</code>
<code>count(element)</code>	返回 <code>element</code> 在列表中出现的次数
<code>extend(newList)</code>	将 <code>newList</code> 的元素插入列表末尾
<code>index(element)</code>	返回 <code>element</code> 在列表中首次出现的索引。如果不在列表中，会产生 <code>ValueError</code> 异常。第 12 章会详细讨论异常问题
<code>insert(index, item)</code>	在位置 <code>index</code> 插入 <code>item</code>
<code>pop([index])</code>	<code>index</code> 参数是可选的。如果无参数调用该方法，会删除并返回列表的最后一个元素。如指定了 <code>index</code> 参数，则删除并返回位置 <code>index</code> 的元素
<code>remove(element)</code>	删除首次在列表中出现的 <code>element</code> 。如果 <code>element</code> 未在列表中，会产生 <code>ValueError</code> 异常
<code>reverse()</code>	当场反转列表内容（不创建一个反转的拷贝）
<code>sort([compare-function])</code>	当场对列表内容排序。可选参数 <code>compare-function</code> 是一个函数，它指定了比较条件。 <code>compare-function</code> 取得列表的任意两个元素（ <code>x</code> 和 <code>y</code> ）。如 <code>x</code> 应出现在 <code>y</code> 之前，返回 -1；如 <code>x</code> 和 <code>y</code> 的顺序无关紧要，返回 0；如 <code>x</code> 应出现在 <code>y</code> 之后，则返回 1。详情参见 5.9 节

图 5.12 列表方法

字典数据类型也提供了多个方法，以便处理存储的数据。图 5.13 演示了 3 个字典方法。第 4~7 行创建字典 `monthsDictionary`，它表示一年中的月份。第 10 行用字典方法 `items` 在屏幕上打印字典的键-值对。方法返回由元组构成的一个列表，每个元组都包含一个键-值对。

```

1 # Fig. 5.13: fig05_13.py
2 # Dictionary methods.
3
4 monthsDictionary = { 1 : "January", 2 : "February", 3 : "March",
5                     4 : "April", 5 : "May", 6 : "June", 7 : "July",
6                     8 : "August", 9 : "September", 10 : "October",
7                     11 : "November", 12 : "December" }
8
9 print "The dictionary items are:"
10 print monthsDictionary.items()
11
12 print "\nThe dictionary keys are:"
13 print monthsDictionary.keys()
14
15 print "\nThe dictionary values are:"
16 print monthsDictionary.values()
17
18 print "\nUsing a for loop to get dictionary items:"
19
20 for key in monthsDictionary.keys():
21     print "monthsDictionary[" + key + "] =", monthsDictionary[ key ]

```

```

The dictionary items are:
[(1, 'January'), (2, 'February'), (3, 'March'), (4, 'April'), (5, 'May'), (6, 'June'), (7, 'July'), (8, 'August'), (9, 'September'), (10, 'October'), (11, 'November'), (12, 'December')]

The dictionary keys are:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

The dictionary values are:
['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December']

Using a for loop to get dictionary items:
monthsDictionary[ 1 ] = January
monthsDictionary[ 2 ] = February
monthsDictionary[ 3 ] = March
monthsDictionary[ 4 ] = April
monthsDictionary[ 5 ] = May
monthsDictionary[ 6 ] = June
monthsDictionary[ 7 ] = July
monthsDictionary[ 8 ] = August
monthsDictionary[ 9 ] = September
monthsDictionary[ 10 ] = October
monthsDictionary[ 11 ] = November
monthsDictionary[ 12 ] = December

```

图 5.13 字典方法 items, keys 和 values

字典方法 `keys` (第 13 行) 返回由字典键构成的一个无序列表。类似地, 字典方法 `values` (第 16 行) 返回字典值的一个无序列表。第 20~21 行演示了字典方法 `keys` 的常见用法。for 循环遍历字典键。每个键都由控制变量 `key` 指派值。第 21 行打印键及其相应的值。图 5.14 总结了各个字典方法。

方法	说明
<code>clear()</code>	从字典删除所有项
<code>copy()</code>	创建并返回字典的一个浅拷贝 (新字典中的元素是对原始字典元素的引用)
<code>get(key [, returnValue])</code>	返回同 <code>key</code> 对应的值。如 <code>key</code> 不在字典中, 同时指定了 <code>returnValue</code> , 就返回指定的值。如果没有指定 <code>returnValue</code> , 就返回 <code>None</code>
<code>has_key(key)</code>	如果 <code>key</code> 在字典中, 就返回 1; 如果不在, 就返回 0
方法	说明
<code>items()</code>	返回一个由元组构成的列表, 每个元组包含一个键-值对
<code>keys()</code>	返回字典中的所有键的一个列表
<code>popitem()</code>	删除任意键-值对, 并作为两个元素的一个元组返回。如字典为空, 会产生 <code>KeyError</code> 异常

方法	说明
	注意, 异常的详情将在第 12 章讨论。该方法尤其适合在从字典中删除一个元素之前 (即打印键-值对) 之前访问
<code>setdefault(key [, dummyValue])</code>	具有与 <code>get</code> 方法类似行为。如果 <code>key</code> 不在字典中, 但同时指定了 <code>dummyValue</code> , 就将键和指定的值插入字典。如果没有指定 <code>dummyValue</code> , 那么值是 <code>None</code>
<code>update(newDictionary)</code>	将来自 <code>newDictionary</code> 的所有键-值对添加到当前字典, 并覆盖同名键的值
<code>values()</code>	返回字典所有值的一个列表
<code>iterkeys()</code>	返回字典键的一个迭代器 (详情在附录 B 讨论)
<code>iteritems()</code>	返回键-值对的一个迭代器 (详情在附录 B 讨论)
<code>itervalues()</code>	返回字典值的一个迭代器 (详情在附录 B 讨论)

图 5.14 字典方法

字典方法 `copy` 可返回一个新字典, 它是原始字典的一个“浅”拷贝。在浅拷贝中, 新字典的元素只是对原始字典的元素的引用。

图 5.15 的交互式会话演示了浅拷贝和深拷贝的差异。首先创建 `dictionary` 字典, 其中包含一个值 (一个数字列表)。然后调用字典方法 `copy` 创建 `dictionary` 的一个浅拷贝, 并将拷贝指派给变量 `shallowCopy`。在两个字典中, 为键“listKey”所存储的值都引用相同的对象。为证明这一点, 我们在 `dictionary` 存储的列表末尾插入值 4。然后, 打印变量 `dictionary` 和 `shallowCopy` 的值。注意在字典的两个拷贝中, 列表都发生了改变。这是创建浅拷贝的必然结果, 因为并没有创建一个完全独立的原始字典的拷贝。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> dictionary = { "listKey" : [ 1, 2, 3 ] }
>>> shallowCopy = dictionary.copy()          # make a shallow copy
>>> dictionary[ "listKey" ].append( 4 )
>>> print dictionary
{'listKey': [1, 2, 3, 4]}
>>> print shallowCopy
{'listKey': [1, 2, 3, 4]}

>>> from copy import deepcopy
>>> deepCopy = deepcopy( dictionary )        # make a deep copy
>>> dictionary[ "listKey" ].append( 5 )
>>> print dictionary
{'listKey': [1, 2, 3, 4, 5]}
>>> print shallowCopy
{'listKey': [1, 2, 3, 4, 5]}
>>> print deepCopy
{'listKey': [1, 2, 3, 4]}
```

图 5.15 浅拷贝和深拷贝的区别

有时, 程序中仅仅使用浅拷贝就足够了, 尤其是假如字典中不包含对其他 Python 对象的引用时 (也就是说, 它们只包含字面意义的数值或者不可变的值)。但有些时候, 却有必要创建一个深拷贝, 令其独立于原始字典。为创建深拷贝, Python 提供了 `copy` 模块。在图 5.15 的交互式会话的剩余部分, 我们创建了变量 `dictionary` 的一个深拷贝。首先从 `copy` 模块导入 `deepcopy` 函数。再调用 `deepcopy`, 并将 `dictionary` 作为参数传给它。函数调用会返回 `dictionary` 的一个深拷贝, 我们把这个拷贝指派给变量 `deepCopy`。现在, 与 `deepCopy["listKey"]` 对应的值独立于与变量 `dictionary` 和 `shallowCopy` 中的键对应的值。为证明这一点, 我们为 `dictionary` 的列表添加一个新值, 并同时打印 `dictionary`、`shallowCopy` 和 `deepCopy` 的值。

浅拷贝和深拷贝反映了 Python 如何对引用 (即对象名称) 进行处理。引用列表和字典这样的对象时, 程序员务必谨慎, 因为更改一个对象, 会影响引用那个对象的所有名称的值。接下来两个小节中, 我们将讨论如何通过向函数传引用来影响对象的值。

软件工程知识 5.2 `deepCopyList = originalList[:]` 进行的是深拷贝, 这意味着 `deepCopyList` 是 `originalList` 的一个深拷贝。

5.7 引用和引用参数

为执行任务，函数需要特定的输入值，这种值是主程序或函数所拥有（或知道）的。主程序（比如一个模拟计数器的程序）可能要求用户输入值，再将那些值发送给函数（比如 `add`, `subtract`）。值（或参数）必须通过特定机制传给函数。在许多程序语言中，都采用两种方式向函数传参数，一种是“传值”，另一种是“传引用”。如参数采取传值方式，会创建参数值的一个拷贝，再将拷贝传给被调用函数。

测试和调试提示 5.3 采用传值，如果对被调用函数的拷贝进行修改，不会影响调用代码中的原始变量值。这有助于避免失误，确保能开发出正确和可靠的软件系统。

如果是传引用，调用者允许被调用函数直接访问调用者的数据，并可对其进行修改。传引用有助于提升性能，因为它避免了拷贝大量数据而产生的开销。但是，传引用不利于安全性，因为被调用的函数能访问调用者的数据。

和其他许多语言不同，传递参数时，Python 不允许程序员选择采用传值还是传引用。Python 参数采用的肯定是“传对象引用”的方式，即函数收到的是对作为参数传递的值的引用。实际上，这种方式相当于传值和传引用的一种综合。如函数收到对一个可变对象（比如字典或列表）的引用，就能修改对象的原始值——相当于通过“传引用”来传递对象。如函数收到对一个不可变对象（比如数字、字符串或者元组）的引用，函数就不能直接修改原始对象——相当于通过“传值”来传递对象。

同样地，程序员必须注意对象有可能被它所传向的一个函数修改。创建大型和复杂的 Python 系统时，务必牢记以上规则，并理解 Python 是怎样对待对象引用的。

5.8 将列表传给函数

本节将进一步讨论引用，研究向函数传递列表时所发生的事情。这儿获得的结论对于其他可变 Python 对象（比如字典）来说也是成立的。要向函数传递一个列表参数，要在不使用方括号的前提下指定列表名称。例如，假定用以下语句创建列表 `hourlyTemperatures`：

```
hourlyTemperatures = [ 39, 43, 45 ]
```

那么以下函数调用：

```
modifyList( hourlyTemperatures )
```

会将 `hourlyTemperatures` 传给函数 `modifyList`。

尽管整个列表可由一个函数更改，但单独的列表元素如果是数值或不可变序列数据类型，则不能修改。要将列表元素传给函数，请在函数调用中将列表元素作为参数使用。

图 5.16 的程序演示了传递整个列表和传递一个列表元素的区别。第 12 行创建了变量 `aList`。第 17~18 行的 `for` 循环打印列表项。第 20 行调用函数 `modifyList`，向函数传递变量 `aList`。函数 `modifyList`（第 4~7 行）将每个元素都乘以 2。为证明 `aList` 的元素已被修改，第 24~25 行的 `for` 循环再次显示列表元素。如输出所示，`aList` 的元素已被 `modifyList` 修改。

```
1 # Fig. 5.16: fig05_16.py
2 # Passing lists and individual list elements to functions.
3
4 def modifyList( aList ):
5
6     for i in range( len( aList ) ):
7         aList[ i ] *= 2
8
9 def modifyElement( element ):
10     element *= 2
11
```



```

12 aList = [ 1, 2, 3, 4, 5 ]
13
14 print "Effects of passing entire list:"
15 print "The values of the original list are:"
16
17 for item in aList:
18     print item,
19
20 modifyList( aList )
21
22 print "\n\nThe values of the modified list are:"
23
24 for item in aList:
25     print item,
26
27 print "\n\nEffects of passing list element:"
28 print "aList[ 3 ] before modifyElement:", aList[ 3 ]
29 modifyElement( aList[ 3 ] )
30 print "aList[ 3 ] after modifyElement:", aList[ 3 ]
31
32 print "\n\nEffects of passing slices of list:"
33 print "aList[ 2:4 ] before modifyList:", aList[ 2:4 ]
34 modifyList( aList[ 2:4 ] )
35 print "aList[ 2:4 ] after modifyList:", aList[ 2:4 ]

```

```

Effects of passing entire list:
The values of the original list are:
1 2 3 4 5

The values of the modified list are:
2 4 6 8 10

Effects of passing list element:
aList[ 3 ] before modifyElement: 8
aList[ 3 ] after modifyElement: 8

Effects of passing slices of list:
aList[ 2:4 ] before modifyList: [6, 8]
aList[ 2:4 ] after modifyList: [6, 8]

```

图 5.16 将列表和单独的列表元素传给函数

第 27~30 行演示了如何向函数传递一个列表元素（即 `aList[3]`，它包含一个数字，但记住数字是不可变的）。程序首先打印 `aList[3]` 的值，即 8。然后，程序调用函数 `modifyElement`（第 9~10 行），向 `element` 参数传递值 8，函数 `modifyElement` 将 `element` 乘以 2。函数终止后，局部变量 `element` 会被清除。在列表中，原始元素 `aList[3]` 的值不会修改，因为 `aList[3]` 的值是不可变的。所以，当控制返回程序的主要部分时，会打印 `aList[3]` 的未修改的值。

分片将创建一个新序列；所以，当程序将一个分片传给函数时，原始序列不受影响。第 33 行在屏幕上打印分片 `aList[2:4]`。第 34 行调用函数 `modifyList`，并传递 `aList[2:4]`。第 35 行会打印调用函数 `modifyList` 的结果——证明原始列表未被修改。

函数 `modifyList` 会遍历它的列表，使用方括号运算符访问元素。如果函数包含以下代码：

```

for item in aList:
    item *= 2

```

列表就会保持不变，因为函数会修改局部变量 `item` 的值，而不修改列表中的特定索引处的值。

5.9 列表排序和搜索

数据排序（即按特定顺序排列数据，比如升序或降序）是一种常见的计算应用。例如，银行需要按账户号码对支票排序，以准备个人月结单。电话公司要先按姓氏对账户排序，再按名字排序，这样能更容易地搜索电话号码。几乎所有单位都要对数据进行排序（而且通常是大量数据）。数据排序是一个有趣

的问题，计算机领域的许多研究都是围绕它而展开的。本节要讨论如何用列表方法 `sort` 对列表排序。

图 5.17 按升序对含 10 个元素的列表 `aList`（第 4 行）排序。第 8~9 行打印列表项。第 11 行调用列表方法 `sort`——它对 `aList` 的元素进行升序排序。程序剩余部分打印排序结果。

```
1 # Fig. 5.17: fig05_17.py
2 # Sorting a list.
3
4 aList = [ 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 ]
5
6 print "Data items in original order"
7
8 for item in aList:
9     print item,
10
11 aList.sort()
12
13 print "\n\nData items after sorting"
14
15 for item in aList:
16     print item,
17
18 print
```

```
Data items in original order
2 6 4 8 10 12 89 68 45 37

Data items after sorting
2 4 6 8 10 12 37 45 68 89
```

图 5.18 列表排序

人们对列表排序进行了大量研究，设计出了许多算法。有的算法易于表示和编程，但效率不够高。有的算法非常复杂，但性能不错。

性能提示 5.1 有时，最简单的算法在性能上也是最差的，它们惟一的优点便是容易编程、测试和调试。为了获得最好的性能，往往需要采取更复杂的算法。

程序员经常需要处理列表中存储的大量数据。可能有必要判断列表中是否包含一个与特定“键值”（key value）匹配的值。在列表中定位特定元素的过程称为“搜索”。

图 5.18 的程序在列表中搜索一个值。第 5 行创建列表 `aList`，其中含有从 0~198（包括 0 和 198 在内）的偶数。第 7 行从用户处获得搜索键（search key），并将值指派给变量 `searchKey`。关键字 `in` 检测列表 `aList` 是否包含用户输入的搜索键（第 9 行）。如果列表含有变量 `searchKey` 中存储的值，第 9 行的表达式就会求值为 `true`；否则为 `false`。

```
1 # Fig. 5.18: fig05_18.py
2 # Searching a list for an integer.
3
4 # Create a list of even integers 0 to 198
5 aList = range( 0, 199, 2 )
6
7 searchKey = int( raw_input( "Enter integer search key: " ) )
8
9 if searchKey in aList:
10     print "Found at index:", aList.index( searchKey )
11 else:
12     print "Value not found"
```

```
Enter integer search key: 36
Found at index: 18

Enter integer search key: 37
Value not found
```

图 5.18 在列表中搜索一个整数

如列表包含搜索键，第10行调用列表方法 `index` 获取搜索键的索引。`index` 取得一个搜索键作为自己的参数，在列表中搜索，并返回与搜索键匹配的列表值的索引。如果列表中没有可与搜索键匹配的任何值，程序将显示一条错误消息。注意，图 5.18 会搜索两次 `aList`（第 9~10 行）；对于较大的序列，这会严重影响性能。为改进性能，程序可直接使用列表方法 `index`，一旦发现参数值不在列表中，就捕捉因此而产生的异常。异常处理问题将在第 12 章详述。

和排序一样，人们对搜索展开了大量研究。除前面介绍的简单搜索，还可采用许多高级搜索算法。

5.10 多下标序列

序列包含的元素也可序列（即列表和元组）。这样的序列有多个下标。多下标序列的常见用途是表示以行、列形式的表格化数据。为标识特定的表格元素，必须指定两个下标——按照约定，第一个标识元素的行，第二个标识元素的列。

用两个下标才能标识一个特定元素的序列称为“双下标序列”或“二维序列”。注意，多下标序列含有的下标可能不止两个。Python 解释器不直接支持多下标序列，但允许程序员在定义单下标元组和列表时，将其元素也指定为单下标元组和列表，由此可获得与多下标序列相同的效果。图 5.19 展示了双下标序列 `a`，其中包含 3 行和 4 列（即 3×4 序列）。通常，含有 m 行、 n 列的序列称为 $m \times n$ 序列。

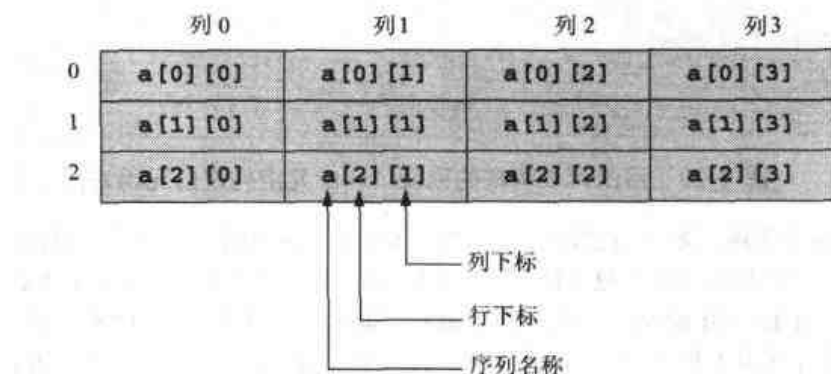


图 5.19 3 行、4 列的双下标序列

在图 5.19 中，序列 `a` 的每个元素都用一个元素名来标识，形式是 `a[i][j]`。其中，`a` 是序列名，`i` 和 `j` 是下标，它们独一无二地标识了 `a` 中每个元素的行和列。注意，对第 1 行（行 0）的元素名，第一个下标全部是 0；对于第 4 列（列 3）的元素名，第二个下标全部是 3。

多下标序列可在创建时初始化，这和单下标序列是类似的。可用以下代码创建一个 2 行 \times 2 列的双下标列表：

```
b = [ [ 1, 2 ], [ 3, 4 ] ]
```

值是按不同的行分组的。第一行是列表第一个元素，第二行是第二个元素。所以，1 和 2 初始化 `b[0][0]` 和 `b[0][1]`；而 3 和 4 初始化 `b[1][0]` 和 `b[1][1]`。多下标序列被作为由序列构成的序列来维护。以下语句：

```
c = ( ( 1, 2 ), ( 3, 4, 5 ) )
```

将创建元组 `c`，它的行 0 包含两个元素（1 和 2），行 1 包含 3 个元素（3，4 和 5）。Python 允许多下标序列的各行具有不同的长度。

图 5.20 演示了如何创建和初始化双下标序列，以及如何用嵌套 `for` 循环遍历序列（即处理序列的每个元素）。

```

1 # Fig. 5.20: fig05_20.py
2 # Making tables using lists of lists and tuples of tuples.
3
4 table1 = [ [ 1, 2, 3 ], [ 4, 5, 6 ] ]
5 table2 = ( ( 1, 2 ), ( 3, ), ( 4, 5, 6 ) )
6
7 print "Values in table1 by row are"
8
9 for row in table1:
10
11     for item in row:
12         print item,
13
14     print
15
16 print "\nValues in table2 by row are"
17
18 for row in table2:
19
20     for item in row:
21         print item,
22
23     print

```

```

Values in table1 by row are
1 2 3
4 5 6

Values in table2 by row are
1 2
3
4 5 6

```

图 5.20 用由列表构成的列表，和由元组构成的元组来创建表格

程序声明了两个序列。第 4 行创建多下标列表 `table1`，并用两个子列表（即列表中的列表）提供 6 个值。序列的第一个子列表（行）包含值 1、2 和 3；第二个子列表包含值 4、5 和 6。

第 5 行创建多下标元组 `table2`，并用 3 个子元组（即元组中的元组）提供 6 个值。第一个子元组（行）包含两个元素，值分别为 1 和 2。第二个子元组包含一个元素，值为 3。第三个元组包含三个元素，值为 4、5 和 6。第 9~14 行使用嵌套 `for` 结构输出列表 `table1` 的行。外层 `for` 结构遍历列表中的行。内层 `for` 结构遍历行中的每个列。程序剩余部分采用类似方式，打印变量 `table2` 的值。

图 5.20 的程序演示了 `for` 结构在处理多下标序列时的重要用处。其他许多常见的序列处理都要用到 `for` 重复结构。例如，以下 `for` 结构可将图 5.19 的序列 `a` 的第 3 行全部元素设为 0：

```

for column in range( len( a[ 2 ] ) ):
    a[ 2 ][ column ] = 0

```

这里说的是“第 3 行”；所以第一个下标肯定是 2（第 1 行是 0，第 2 行是 1）。`for` 结构只改变第二个下标（即列下标）。上述 `for` 结构等价于以下赋值语句：

```

a[ 2 ][ 0 ] = 0
a[ 2 ][ 1 ] = 0
a[ 2 ][ 2 ] = 0
a[ 2 ][ 3 ] = 0

```

以下 `for` 嵌套结构可计算序列 `a` 所有元素的总和：

```

total = 0

for row in a:
    for column in row:
        total += column

```

`for` 结构每次计算一行的总和。外层 `for` 结构遍历表格中所有行，使每行的元素都能由内层的 `for` 结

构求和。嵌套 for 结构终止后，就能显示出 total 值。

图 5.21 的程序针对 3×4 列表 grades 执行其他几种常见的序列处理。列表每一行都代表一名学生，每一列都代表学生在本学期参加的三门考试之一的成绩。由 4 个函数来处理列表。函数 printGrades（第 5~25 行）以表格形式打印存储在列表 grades 中的数据。函数 minimum（第 28~38 行）判断任何学生在本学期的最低成绩。函数 maximum（第 41~51 行）判断任何学生在本学期的最高成绩。函数 average（第 54~60 行）确定了特定学生的学期平均成绩。注意，第 55 行将 total 初始化成 0.0，使函数能返回一个浮点值。

```

1 # Fig. 5.21: fig05_21.py
2 # Double-subscripted list example.
3
4
5 def printGrades( grades ):
6     students = len( grades )      # number of students
7     exams = len( grades[ 0 ] )   # number of exams
8
9     # print table headers
10    print "The list is:"
11    print "      ",
12
13    for i in range( exams ):
14        print "[%d]" % i,
15
16    print
17
18    # print scores, by row
19    for i in range( students ):
20        print "grades[%d] " % i,
21
22        for j in range( exams ):
23            print grades[ i ][ j ], "",
24
25    print
26
27
28 def minimum( grades ):
29     lowScore = 100
30
31     for studentExams in grades:    # loop over students
32
33         for score in studentExams: # loop over scores
34
35             if score < lowScore:
36                 lowScore = score
37
38     return lowScore
39
40
41 def maximum( grades ):
42     highScore = 0
43
44     for studentExams in grades:    # loop over students
45
46         for score in studentExams: # loop over scores
47
48             if score > highScore:
49                 highScore = score
50
51     return highScore
52
53
54 def average( setOfGrades ):
55     total = 0.0
56
57     for grade in setOfGrades:      # loop over student's scores
58         total += grade
59

```

```

60     return total / len( setOfGrades )
61
62
63 # main program
64 grades = [ [ 77, 68, 86, 73 ],
65            [ 96, 87, 89, 81 ],
66            [ 70, 90, 86, 81 ] ]
67
68 printGrades( grades )
69 print "\n\nLowest grade:", minimum( grades )
70 print "Highest grade:", maximum( grades )
71 print "\n"
72
73 # print average for each student
74 for i in range( len( grades ) ):
75     print "Average for student", i, "is", average( grades[ i ] )

```

```

The list is:
[0] [1] [2] [3]
grades[0] 77 68 86 73
grades[1] 96 87 89 81
grades[2] 70 90 86 81

Lowest grade: 68
Highest grade: 96

Average for student 0 is 76.0
Average for student 1 is 88.25
Average for student 2 is 81.75

```

图 5.21 双下标列表

函数 `printGrades` 使用列表 `grades` 以及变量 `students` (列表中的行数) 和 `exams` (列表中的列数)。函数遍历列表 `grades`, 通过嵌套 `for` 结构以表格方式打印成绩。外层 `for` 结构 (第 19~25 行) 遍历 `i` (即行下标), 内层 `for` 结构 (第 22~23 行) 遍历 `j` (即列下标)。

函数 `minimum` 和 `maximum` 使用嵌套 `for` 结构遍历列表 `grades`。其中, `minimum` 将每个成绩与变量 `lowScore` 比较。如成绩小于 `lowScore`, 就将 `lowScore` 设为那个成绩 (第 36 行)。嵌套结构执行完毕后, `lowScore` 就包含了双下标列表中的最小成绩。函数 `maximum` 的原理类似于 `minimum`。

函数 `average` 要取得一个参数, 即一个单下标列表, 它由特定学生的考试成绩构成。第 75 行调用 `average` 时, 参数是 `grades[i]`, 它的意思是 `f` 将双下标列表 `grades` 的一个特定的行传给 `average`。例如, 参数 `grades[1]` 表示表存储在 `grades` 列表第 2 行的 4 个值 (即成绩的一个单下标列表)。记住在 Python 中, 双下标列表的元素就是单下标列表。函数 `average` 会计算列表元素之和, 结果除以考试结果的数量, 并返回浮点数形式的平均成绩。

上例演示了如何使用双下标列表。但是, 如需计算纯数字问题 (即多维数组), 基本的 Python 语言将无法有效地处理它们。此时, 应用一个名为 NumPy 的包。NumPy (即“数字 Python”) 提供了丰富的数组处理模块, 并提供了多维数组对象, 以进行高效的计算。欲知 NumPy 的详情, 请访问 sourceforge.net/projects/numpy。

第 2~5 章介绍了 Python 的基本编程技术。第 6 章将利用这些技术设计基于 Web 的应用程序。第 7~9 章将介绍面向对象编程技术, 以便在本书后半部分构建复杂的应用程序。

第 6 章 公共网关接口 (CGI) 入门

学习目标

- 理解 CGI 协议
- 理解 HTTP
- 实现 CGI 脚本
- 用 XHTML 表单向 CGI 脚本发送信息
- 理解和解析查询字符串
- 用 cgi 模块处理来自 XHTML 表单的信息

6.1 概述

“公共网关接口”(Common Gateway Interface, CGI)描述了一系列协议,应用程序(通常称为 CGI 程序或 CGI 脚本)可利用这些协议直接与 Web 服务器交互,并间接与 Web 浏览器(即客户程序)交互。Web 服务器是一种特殊的软件应用程序,它通过提供资源(比如网页)来响应客户应用程序的请求。CGI 协议通常动态生成网页。如果 Web 服务器上的一个程序在用户每次请求网页时生成那个页的内容,就可以认为该网页是动态的。例如,网页中的一个表单可要求用户输入邮政编码。输入并提交邮政编码后,Web 服务器可用一个 CGI 程序创建网页,显示用户所在地区的天气信息。相反,静态网页内容从来不变,除非 Web 开发者亲自编辑它。

CGI 之所以是“公共”的,是因为它不依赖于任何操作系统(无论 Linux 还是 Windows),也不依赖于任何程序语言或任何 Web 服务器软件。CGI 可用于几乎任何程序语言或脚本语言,比如 C, Perl 和 Python。本章解释了 Web 客户与服务器如何交互,介绍了 CGI 的基础知识,并讲述了如何用 Python 编写 CGI 脚本。

CGI 协议最早是在 1993 年由 NCSA(美国国家超级计算机应用中心——www.ncsa.uiuc.edu)开发的。当时, NCSA 打算通过 CGI 来支持一种名为“HTTPd”的 Web 服务器,并将 CGI 设计为生成动态 Web 内容的一种简单工具。它简洁、高效的语法,使其迅速成为全球通用的非正式协议。其他 Web 服务器相继提供了对 CGI 的支持,其中包括 Microsoft Internet 信息服务(IIS)和 Apache(www.apache.org)。

6.2 客户和 Web 服务器交互

本节讨论 Web 服务器和客户应用程序的交互。网页(Web Page)采用其最简单的形式,要么是一个 HTML 文档,要么是一个 XHTML 文档(本章使用后者)。XHTML 文档是纯文本文件,其中包含用于描述文档在 Web 浏览器上如何显示的标记。比如以下 XHTML 标记:

```
<title>My Web Page</title>
```

它指出开始标记<title>和结束标记</title>之间的文本是网页“标题”。浏览器会按特定格式呈现这些标记之间的文本。

XHTML 要求语法正确的文档——标记必须遵循特定的规则。例如, XHTML 标记必须全部采用小写字母,所有开始标记都必须有对应的结束标记。

每个网页都有对应的 URL(统一资源定位符),这是它的地址。URL 包含的信息可将浏览器定向到用户希望访问的资源。比如以下 URL:

```
http://www.deitel.com/books/downloads.html
```

它的第一部分是 `http://`，它指明资源要通过 HTTP（超文本传输协议）来获取。交互期间，Web 服务器和客户端通过与平台无关的 HTTP 协议来通信，该协议允许在 Internet 上（比如在 Web 服务器和浏览器之间）发送请求和发送文件。6.2.3 节会具体讨论 HTTP。

URL 的下一个部分是 `www.deitel.com`，它代表服务器的主机名，即服务器计算机的名称，所需的资源就位于该计算机上。DNS（域名系统）服务器会将主机名 `www.deitel.com` 转换成相应的 IP（网际协议）地址，例如 `207.60.134.230`，它在 Internet 上独一无二地标识了服务器计算机（就像电话号码独一无二地标识一条电话线一样）。这个转换过程称为“DNS 搜索”。DNS 服务器维护着主机名及其对应 IP 地址的一个数据库。

URL 剩余的部分指定了请求的资源（`/books/downloads.html`）。URL 的这个部分指定了资源的名称（`downloads.html`）及其路径（`/books`）。Web 服务器将 URL 映射至服务器上的一个文件（或其他资源，比如 CGI 程序），或映射至服务器所在网络的另一个资源。然后，Web 服务器将请求的文档返回客户端。路径代表 Web 服务器文件系统上的一个目录。资源也可能是动态创建的，不存在于服务器计算机的任何地方。在这种情况下，URL 用主机名来定位正确的服务器，服务器则根据路径和资源信息定位（或创建）资源，以响应客户端的请求。如后文所述，URL 还可为服务器上的 CGI 程序提供输入。

6.2.1 系统体系结构

Web 服务器通常是一个“多层应用程序”的一部分，有时也将其称为 n 层应用程序。多层应用程序将功能分隔到单独的“层”（即逻辑性的功能分组）。不同的层可位于同一台计算机上，也可位于多台计算机上。图 6.1 展示了一个 3 层应用程序的基本结构。

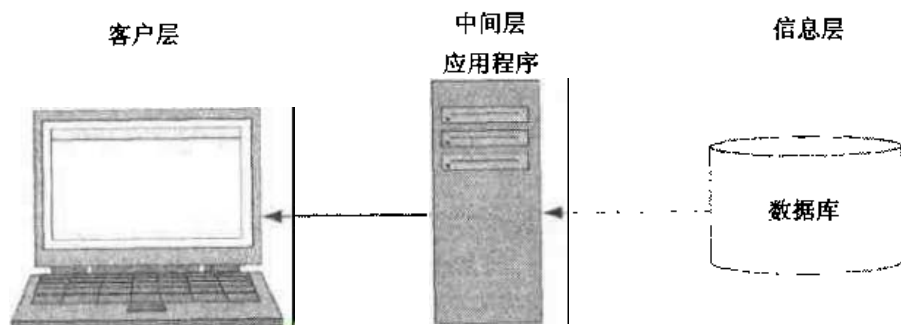


图 6.1 3 层应用程序模型

其中，“信息层”（也称为数据层或底层）维护着应用程序所需的数据。这个层通常将数据存储到一个关系数据库管理系统（RDBMS）中。RDBMS 的详情将在第 17 章介绍。例如，一家零售商店可用数据库来存储产品信息，包括产品说明、价格以及库存数量等等。同一个数据库还可存储客户信息，包括用户名、账单地址和信用卡号码等等。

“中间层”实现了事务处理逻辑和表示逻辑，以控制客户端和数据之间的交互。中间层相当于数据和客户端的中间人。其中，中间层的事务处理逻辑处理来自客户层的请求（例如查看产品目录的请求），并从数据库获取数据。中间层的表示逻辑则处理来自信息层的数据，并以特定方式将内容呈现给客户。

事务处理逻辑用于强制“事务处理规则”或“商业规则”，并先确保数据是可靠的，再实际更新数据库或者向客户端展示数据。事务处理规则（商业规则）规定了客户端怎样访问应用程序数据，以及应用程序如何处理数据。

中间层还实现了应用程序的表示逻辑。Web 应用程序通常以 XHTML 文档的形式向客户端展示信息（较早的应用程序则将信息呈现为 HTML 形式）。许多 Web 应用程序都以 WML（无线标记语言）文档形式向无线客户端展示信息。WML 的详情将在第 23 章讨论。

最后,“客户层”(也称为顶层)是应用程序的用户界面。用户通过用户界面同应用程序交互。这实际会导致客户端与中间层交互,以便向信息层发出请求,以及从中获取数据。最后,客户端向用户显示从中间层获取的数据。

6.2.2 访问 Web 服务器

要从 Web 服务器请求文档,用户必须知道 Web 服务器软件所在的机器的名称(名为“主机名”)。用户可从本地 Web 服务器(它们在用户自己的机器上)请求文档,也可从远程 Web 服务器(它们在不同的机器上)请求文档。

可通过机器名或 localhost(引用本地机器的一个主机名)从本地 Web 服务器请求文档。本书使用的是 localhost。在 Windows 98 中,要想知道机器名,请右击【网上邻居】,从弹出菜单中选择【属性】,打开【网络】对话框,再单击【标识】选项卡。计算机名会显示在【计算机名:】区域。单击【取消】关闭对话框。在 Windows 2000 中,右击【网上邻居】,选择【属性】,打开【网络和拨号连接】文件夹,单击【网络标识】链接。随后在【系统特性】窗口的【完整的计算机名称:】区域,显示了计算机名。在 Windows XP 中,右击【我的电脑】,再单击【计算机名】选项卡,计算机名将显示在【完整的计算机名称:】区域。在大多数 Linux 机器上,在 shell 提示符中输入命令 hostname,即可看到计算机名。

客户要想访问服务器,可指定服务器的域名或 IP 地址(例如在 Web 浏览器的【地址】区域输入)。域名代表 Internet 上的一组主机;它与主机名(比如 www——Web 服务器使用的一个常见主机名)和顶级域名(TLD)一起构成了“完全合格的主机名”。这样一来,就可采用一种对用户友好的方式来标识 Internet 上的站点。在完全合格主机名中,TLD 通常描述拥有域名的组织类型。例如,com 这个 TLD 通常代表商业组织,而 org 通常代表非赢利组织。此外,每个国家都有自己的 TLD,比如 cn 代表中国,et 代表埃塞俄比亚,om 代表阿曼,us 代表美国。

6.2.3 HTTP 事务处理

在探索 CGI 的工作原理之前,有必要先了解联网和万维网。本节将从技术角度讨论浏览器如何与 Web 服务器交互以显示网页,并介绍超文本传输协议(HTTP)。此外,还要讨论 HTTP 的特定组件,它们使客户和服务端能采取统一的、可预测的方式交互及交换信息。

HTTP 请求通常将数据投送给一个“服务器端表单处理程序”,由后者处理数据。例如,假定用户参加一次网上调查,Web 服务器将接收用户在 XHTML 表单中输入的、作为请求的一部分的信息。

用户输入 URL 时,客户端必须发出资源请求。两种最常见的 HTTP 请求类型(也称为请求方法)是 get 和 post。这些请求类型从 Web 服务器获取资源,并将客户表单数据发送给 Web 服务器。get 请求将表单内容作为 URL 的一部分发送。比如在以下 URL 中:

```
www.somesite.com/search?query=value
```

问号(?)之后的信息(query=value)代表用户输入。例如,假定要搜索“Massachusetts”,URL 的最后部分就可能是?query=Massachusetts。大多数 Web 服务器都将 get 请求查询字符串限制在 1024 个字符以内。如果查询字符串超出这一限制,就必须使用 post 请求。post 请求中发送的数据不是 URL 的一部分,用户看不见它们。如果表单包含许多字段,那就通常由 post 请求进行提交。一些敏感的表单字段,比如密码,也通常使用这种请求类型来发送。

为发出请求,浏览器要向服务器发送一条 HTTP 请求消息(图 6.2,步骤 1)。HTTP 支持两种请求类型,即 get 和 post。get 请求如果采用它的最简单的形式,那么格式为:GET /books/downloads.html HTTP/1.1。其中,单词 GET 称为“HTTP 方法”,指出客户要请求一个资源。请求的下一个部分提供了资源(一个 HTML/XHTML 文档)的名称(即 downloads.html)以及路径(即/books/)。请求的最后一部分提供了协议名称和版本号(即 HTTP/1.1)。

理解 HTTP 1.1 的服务器会翻译请求，并进行响应（参见图 6.2 - 步骤 2）。服务器响应的是一行消息，首先是 HTTP 版本，后续数值状态码和一条短语，指明事务处理状态。例如：

```
HTTP/1.1 200 OK
```

表明成功，而以下短语：

```
HTTP/1.1 404 Not Found
```

则表明在服务器上 URL 所指定的位置处，未找到请求的资源。

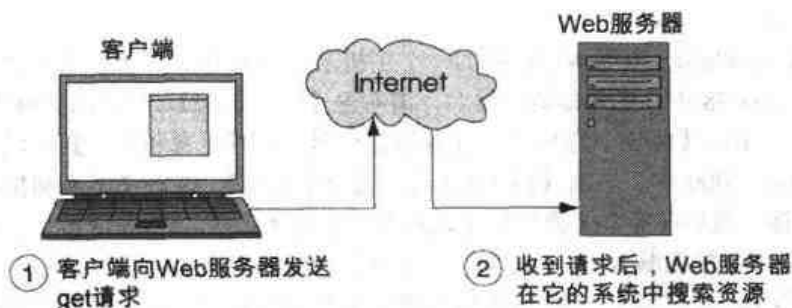


图 6.2 客户同 Web 服务器交互 - 步骤 1：发出请求 - GET /books/downloads.html HTTP/1.1

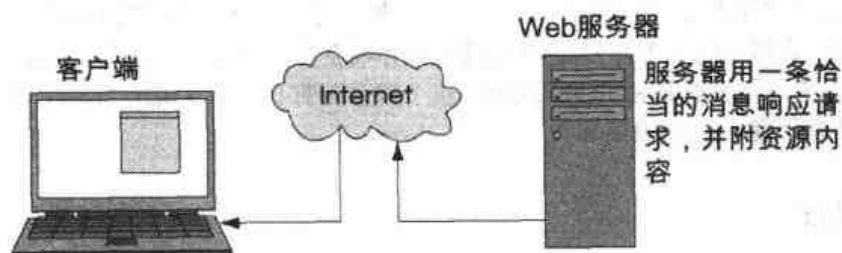


图 6.2 客户同 Web 服务器交互 - 步骤 2：HTTP 响应 - HTTP/1.1 200 OK

浏览器可能“缓存”网页（将网页保存到本地磁盘），以加快重载速度，减少浏览器需要下载的数据量。然而，浏览器通常不会缓存服务器对 post 请求的响应，因为后续的 post 请求可能不包含相同的信息。例如，参加网上调查的几个用户可能请求相同的网页。每个用户的响应都可能改变总体调查结果，从而改变 Web 服务器上的数据。

另一方面，Web 浏览器通常会缓存服务器对 get 请求的响应。使用搜索引擎时，get 请求要向其提交在 XHTML 表单中指定的搜索条件。然后，搜索引擎执行搜索，并以网页形式返回结果。这些网页会缓存下来，用户以后执行同样的搜索时，就直接使用缓存的版本。

服务器通常会发送一个或多个“HTTP 标头”，它们提供了与响应的数据有关的更多信息。在这个例子中，服务器发送的是一个 HTML/XHTML 文本文档，所以 HTTP 标头是：

```
Content-type: text/html
```

这种信息称为内容的 MIME（多用途 Internet 邮件扩展）类型。MIME 是一项 Internet 标准，规定了消息应如何格式化，客户端要根据内容类型来决定如何向用户显示内容。发送的每种数据都有对应的 MIME 类型，帮助浏览器决定如何处理它收到的数据。例如，text/plain 这一 MIME 类型指出是纯文本内容，显示时不能将任何内容解释成 HTML 或 XHTML 标记。类似地，image/gif 指出内容是 GIF（图形交换格式）图片。浏览器收到这种内容时，会尝试显示图片。欲知 MIME 的详情，请访问 www.nacs.uci.edu/indiv/ehood/MIME/MIME.html。

HTTP 标头之后是一个空行（回车符、换行符或两者的组合），它向客户指出服务器已完成 HTTP

标头的发送。然后，服务器发送在请求的 HTML/XHTML 文档 (downloads.html) 中的文本。一旦资源传输结束，连接将终止。客户端的浏览器会解释它收到的文本，并显示（或呈现）结果。

本节讨论了 Web 浏览器（比如 Microsoft Internet Explorer 或 Netscape Communicator）和 Web 服务器（比如 Apache 或 IIS）之间执行的简单 HTTP 事务处理。下一小节将介绍 CGI 编程。

6.3 简单的 CGI 脚本

Web 应用程序采用两类脚本编程：服务器端和客户端。CGI 脚本是服务器端脚本的一个例子，因其在服务器上运行。使用服务器端脚本，程序员对网页能有更大的控制，因为这种脚本可操纵数据库和其他服务器资源。客户端脚本编程的一个例子是 JavaScript。客户端脚本可使用浏览器的功能、处理浏览器文档以及校验用户输入等等。

服务器上执行的脚本通常为客户生成自定义的响应。例如，客户可连接航空公司的 Web 服务器，并请求 9 月 19 日到 11 月 5 日之间从波士顿到圣安东尼奥的所有航班的列表。服务器要查询数据库，动态生成 XHTML 内容，列出所有航班，并将 XHTML 文档发送给客户。利用这种技术，客户可连接航空公司的 Web 服务器，从数据库中获得最新航班信息。

相较于客户端脚本语言，服务器端脚本语言提供更广泛的编程能力。例如，服务器端脚本可访问服务器的目录结构，而客户端脚本无此权限。

服务器端脚本还可访问服务器端软件，以扩展服务器的功能。对于 Microsoft Web 服务器，这些软件称为“COM 组件”；对于 Apache Web 服务器，则称为“模块”。组件和模块的范围很广，从程序语言支持，一直到网页访问计数器。

软件工程知识 6.1 服务器端脚本对客户是不可见的，客户只能看见服务器发送的内容。

只要服务器上的文件保持不变，每次访问时，它对应的 URL 总会在客户的浏览器中显示同样的内容。要想改变文件内容（比如添加新链接或最新的公司新闻），必须人工更改文件（可使用文本编辑器或者网页设计软件），再将改动过的文件存回服务器。

如果想创建有趣和动态的网页，人工修改网页是不切实际的。例如，假如希望网页总是显示最新的日期或天气，就需要连续更新网页。

本章的例子主要依赖 XHTML 和层叠样式表 (CSS)。CSS 允许文档作者独立于文档的结构（段落标题、正文、链接等等），指定元素在网页上的表示方式（间距和页边距等）。

图 6.3 展示了我们第一个 CGI 脚本的完整源代码。其中，第 1 行：

```
#!c:\Python\python.exe
```

是一个预编译指令（有时称为 pound-bang 或 sh-bang），指定了 Python 解释器在服务器上的位置。它必须是 CGI 脚本的第一行。本章的例子是为 Windows 用户写的。如果是 UNIX 或 Linux 机器，预编译指令可能是：

```
#!/usr/bin/python
#!/usr/local/bin/python
#!/usr/bin/env python
```

具体由 Python 解释器的位置决定。如果不知道 Python 解释器的位置，请与服务器管理员联系。

常见编程错误 6.1 运行脚本的 Web 服务器不理解.py 扩展名时，忘记在 CGI 脚本第一行添加预编译指令（#!），是错误的。

```
1 #!c:\Python\python.exe
2 # Fig. 6.3: fig06_03.py
3 # Displays current date and time in Web browser.
4
```

```

5 import time
6
7 def printHeader( title ):
8     print """Content-type: text/html
9
10 <?xml version = "1.0" encoding = "UTF-8"?>
11 <!DOCTYPE html PUBLIC
12     "-//W3C//DTD XHTML 1.0 Strict//EN"
13     "DTD/xhtml11-strict.dtd">
14 <html xmlns = "http://www.w3.org/1999/xhtml">
15 <head><title>%s</title></head>
16
17 <body>""" % title
18
19 printHeader( "Current date and time" )
20 print time.ctime( time.time() )
21 print "</body></html>"

```

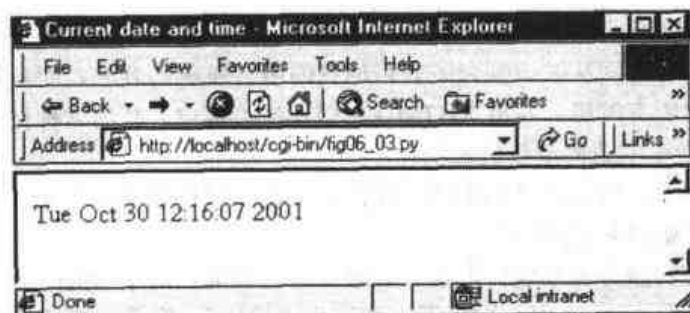


图 6.3 显示日期和时间的 CGI 脚本

第 5 行导入 `time` 模块 (`import time`)。该模块包含了 Web 服务器的当前时间，并在用户浏览器中显示。第 7~17 行定义 `printHeader` 函数，它取得与网页标题对应的参数 `title`。第 8 行打印 HTTP 标头。注意，第 9 行是一个空行，标明 HTTP 标头结束。最后一个 HTTP 标头之后必须是一个空行，否则 Web 浏览器无法正确显示内容。第 10~14 行打印 XML 声明、文档类型声明和开始标记 `<html>`。XML 的详情参见第 15 章。第 15~17 行包含 XHTML 文档头和标题，并开始 XHTML 文档主体。

常见编程错误 6.2 忘记在 HTTP 标头之后添加空行是错误的。

第 19 行开始程序的主要部分，它调用函数 `printHeader`，并传递一个参数以表示网页标题。第 20 行调用 `time` 模块中的两个函数以打印当前时间。其中，函数 `time.time` 返回一个浮点值，它代表自 1970 年 1 月 1 日午夜之后流逝的秒数。函数 `time.ctime` 则取得这个秒数作为自己的参数，返回符合阅读习惯的字符串以表示当前时间。最后，程序打印 XHTML 主体，并添加文档结束标记。要想了解 `time` 模块中的所有函数，请访问：

www.python.org/doc/current/lib/module-time.html

注意，程序几乎完全由 `print` 语句构成。到目前为止，`print` 的输出都是在屏幕上显示的。但是，从技术角度说，`print` 的目标是“标准输出”——由应用程序提供给用户的一个信息流。通常，标准输出在屏幕上显示，但也可发送给打印机，写到一个文件等等。Python 程序作为一个 CGI 脚本执行时，服务器将标准重定向至客户端的 Web 浏览器。浏览器对头和标记进行解释，把它们当做是服务器对 XHTML 文档请求的一个普通响应。

程序要想执行，必须正确配置服务器。注意，本书使用的是 Apache Web 服务器。要详细了解如何获得和配置 Apache，请参考我们在 www.dittel.com 公布的 Python Web 资源。一旦服务器可用，Web 站点管理员就需要指定 CGI 脚本位于何处，以及允许它们使用什么名字。在我们的例子中，Python 文件放在 Web 服务器的 `cgi-bin` 目录。对于 UNIX 和 Linux 用户，还需要先设置适当的权限，然后才能执行程序。例如以下 UNIX 和 Linux 命令：

```
chmod 755 fig06_02.py
```

可为客户授予读取和执行 fig06_02.py 的权限。

假定服务器位于本地计算机上，为执行程序，可在浏览器的“地址”栏中输入：

```
http://localhost/cgi-bin/fig06_02.py
```

如果服务器位于不同计算机上，请将 localhost 替换成服务器的主机名或 IP 地址。注意，localhost 的 IP 地址肯定是 127.0.0.1。请求文档会导致服务器执行程序，并返回结果。

图 6.4 展示了调用 CGI 脚本的过程。首先（步骤 1），客户从服务器请求名为 fig06_02.py 的资源，这类似于在上一个例子中由客户请求 download.html。注意。如果服务器尚未配置好，无法正确处理 CGI 脚本，就有可能将 Python 代码作为普通文本返回给客户。

然而，正确配置的 Web 服务器能辨别出特定的资源需要进行不同的处理。例如，假定资源是 CGI 脚本，脚本必须由 Web 服务器执行。资源要想标识为 CGI 脚本，可采取两种方式——要么采用特殊的文件扩展名（比如.cgi 或.py），要么存储到一个特定的目录（通常是 cgi-bin）。此外，服务器管理员必须明确指定远程访问和 CGI 脚本执行权限。

服务器分辨出资源是一个 Python 脚本，并调用 Python 来执行脚本（步骤 2）。程序执行之后，发送给标准输出的文本会返回给 Web 服务器（步骤 3）。最后，Web 服务器在输出中打印附加的一行，指出 HTTP 事务处理的状态（比如 HTTP/1.1 200 OK 表明成功），再将完整的文本主体发送给客户（步骤 4）。

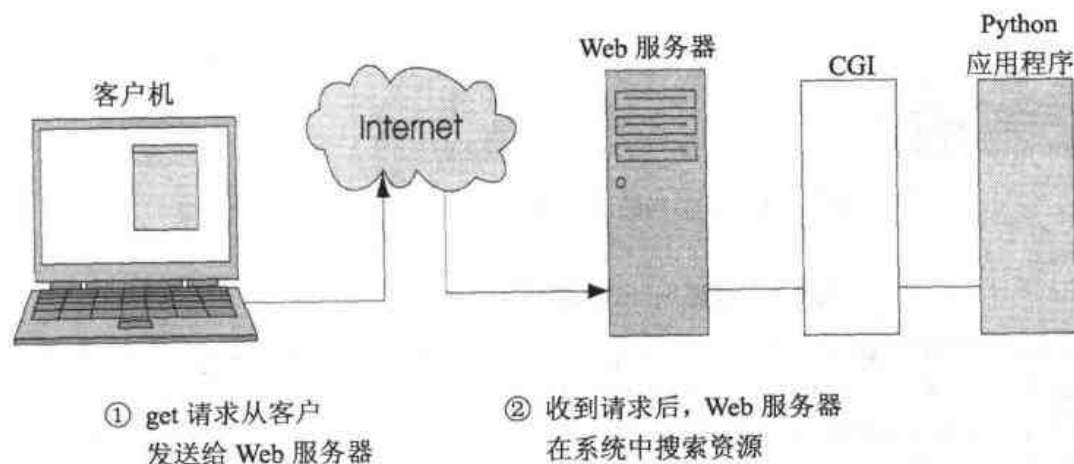


图 6.4 步骤 1: GET 请求, GET /cgi-bin/fig06_02.py HTTP/1.1

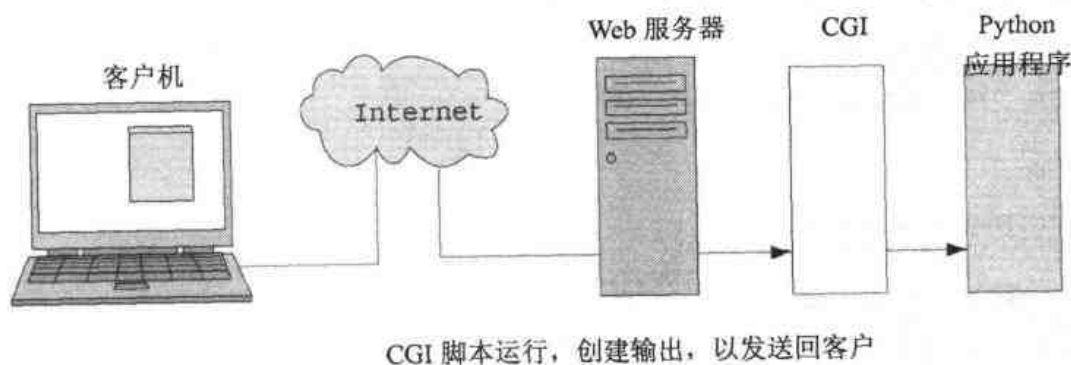


图 6.4 步骤 2: Web 服务器开始执行 CGI 脚本

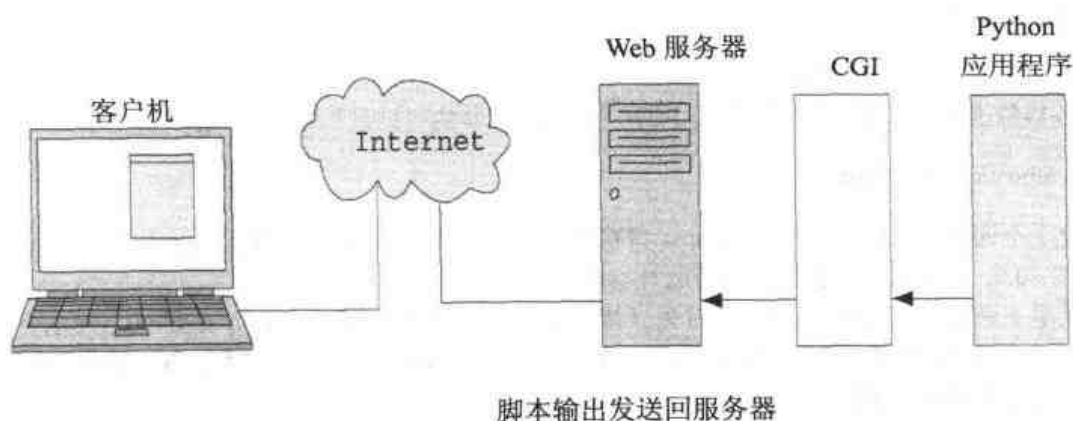


图 6.4 步骤 3: 脚本输出发送给 Web 服务器

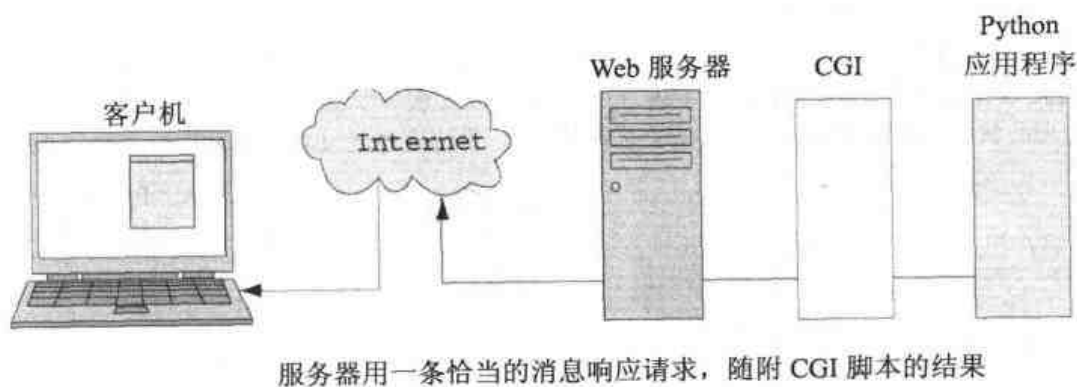


图 6.4 步骤 4: HTTP 响应, HTTP/1.1 200 OK

客户端浏览器处理 XHTML 输出，并显示结果。要注意的是，浏览器并不知道服务器上为执行 CGI 脚本和返回 XHTML 输出所做的工作。浏览器像平常那样请求一个资源，也像平常那样收到一个响应。客户端不需要安装 Python 解释器，因为脚本是在服务器上执行的。客户端只需接收和处理脚本的输出。

现在来研究一个更复杂的 CGI 程序。图 6.5 将所有 CGI 环境变量及其对应的值组织到一个 XHTML 表格中，然后在 Web 浏览器上显示。环境变量包含了同脚本执行环境有关的信息。其中包括当前用户名以及操作系统的名称。CGI 程序用环境变量获得同客户有关的信息，比如客户的 IP 地址、操作系统类型、浏览器类型等等，还可获得由客户传递给 CGI 程序的信息。

```

1  #!c:\Python\python.exe
2  # Fig. 6.5: fig06_05.py
3  # Program displaying CGI environment variables.
4
5  import os
6  import cgi
7
8  def printHeader( title ):
9      print """Content-type: text/html
10
11  <?xml version = "1.0" encoding = "UTF-8"?>
12  <!DOCTYPE html PUBLIC
13      "-//W3C//DTD XHTML 1.0 Strict//EN"
14      "DTD/xhtml1-strict.dtd">
15  <html xmlns = "http://www.w3.org/1999/xhtml">
16  <head><title>%s</title></head>
17
18  <body>""" % title

```

```

19
20 rowNumber = 0
21 backgroundColor = "white"
22
23 printHeader( "Environment Variables" )
24 print """<table style = "border: 0">"""
25
26 # print table of cgi variables and values
27 for item in os.environ.keys():
28     rowNumber += 1
29
30     if rowNumber % 2 == 0:          # even row numbers are white
31         backgroundColor = "white"
32     else:                          # odd row numbers are grey
33         backgroundColor = "lightgrey"
34
35     print """<tr style = "background-color: %s">
36     <td>%s</td><td>%s</td></tr>""" % ( backgroundColor,
37         cgi.escape( item ), cgi.escape( os.environ[ item ] ) )
38
39 print """</table></body></html>"""

```

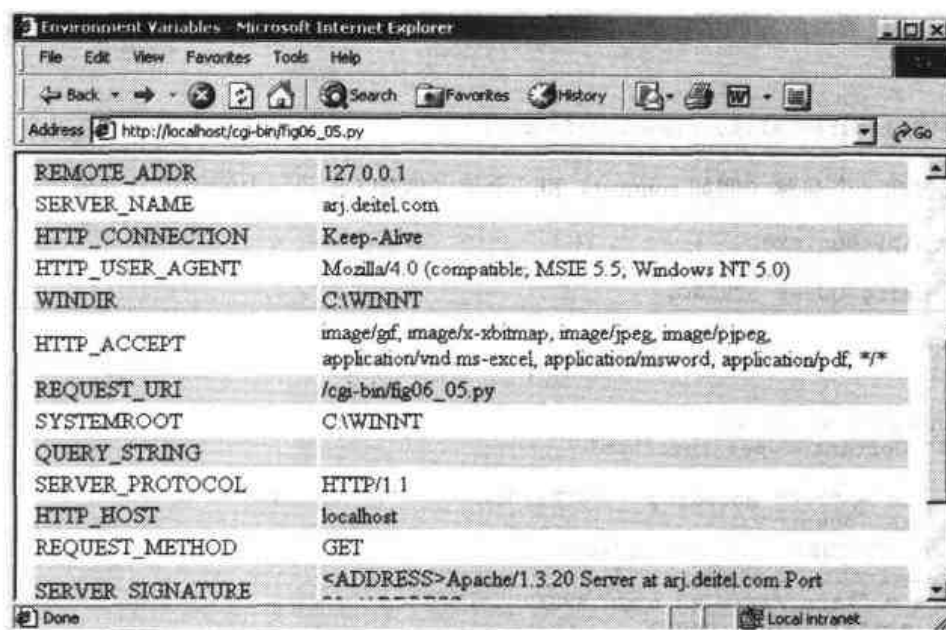


图 6.5 用于显示环境变量的 CGI 程序

第 6 行导入 `cgi` 模块。该模块提供了几项同 CGI 有关的功能，包括文本格式化、表单处理和 URL 解析。本例用 `cgi` 模块格式化 XHTML 文本；以后的例子要用 `cgi` 模块处理 XHTML 表单。

第 8~18 行定义函数 `printHeader`，它和前例定义的函数一模一样。主程序打印包含环境变量的 XHTML 表格（第 24~39 行）。`os.environ` 数据成员容纳了所有环境变量（第 27 行）。该数据成员的行为就像一个字典，因此，可通过 `keys` 方法访问它的键，并用 `[]` 运算符访问它的值。第 30~33 行设置每一行的背景颜色。针对每个环境变量，第 35~37 行创建一个新的表格行，其中包含了键及其对应的值。

注意，第 37 行调用函数 `cgi.escape`，并将每个环境变量名和值作为参数传给它。该函数取得一个字符串，并返回正确格式化的 XHTML 字符串。所谓“正确格式化”，意味着特殊 XHTML 字符（比如小于符号和大于符号）会被“转义”。例如，函数 `escape` 会将“<”替换成“<”，将“>”替换成“>”，并将“&”替换成“&”。这样一来，浏览器会显示普通字符，而不是将字符视为标记。打印所有环境变量后，就结束标记 `table`、`body` 和 `html`（第 39 行）。

6.4 向 CGI 脚本发送输入

前面介绍了一个处理预设环境变量的 CGI 脚本。现在用一个环境变量向 CGI 脚本提供数据（例如客户名和搜索引擎查询）。本节介绍的是环境变量 `QUERY_STRING`，它可帮助我们做到这一点。`QUERY_STRING` 变量包含了在 GET 请求中附加到 URL 末尾的信息（位于一个问号之后）。例如以下 URL：

`www.somesite.com/cgi-bin/script.py?state=California`

会导致 Web 浏览器从 `www.somesite.com` 请求一个资源。资源要通过执行 CGI 脚本（`cgi-bin/script.py`）来生成。问号（?）之后的信息（`state=California`）由 Web 服务器指派给 `QUERY_STRING` 环境变量。注意，问号本身并非所请求的资源的一部分，也不是查询字符串的一部分，它只是资源和查询字符串之间的一个定界符（或分隔符）。

图 6.6 展示了一个简单的 CGI 脚本，它读取并响应通过 `QUERY_STRING` 环境变量传递的数据。CGI 脚本读取解释格式化数据所需的字符串。在这个例子中，查询字符串包含一系列用符号 `&` 分隔的名-值对，例如：

`country=USA&state=California&city=Sacramento`

每个名-值对都包括一个名称（例如 `country`）和一个值（例如 `USA`），中间用等号分隔。

```
1  #!c:\Python\python.exe
2  # Fig. 6.6: fig06_06.py
3  # Example using QUERY_STRING.
4
5  import os
6  import cgi
7
8  def printHeader( title ):
9      print """Content-type: text/html
10
11  <?xml version = "1.0" encoding = "UTF-8"?>
12  <!DOCTYPE html PUBLIC
13      "-//W3C//DTD XHTML 1.0 Strict//EN"
14      "DTD/xhtml1-strict.dtd">
15  <html xmlns = "http://www.w3.org/1999/xhtml">
16  <head><title>%s</title></head>
17
18  <body>""" % title
19
20  printHeader( "QUERY_STRING example" )
21  print "<h1>Name/Value Pairs</h1>"
22
23  query = os.environ[ "QUERY_STRING" ]
24
25  if len( query ) == 0:
26      print """<p><br />
27          Please add some name-value pairs to the URL above.
28          Or try
29          <a href = "fig06_06.py?name=Veronica&age=23">this</a>.
30          </p>"""
31  else:
32      print """<p style = "font-style: italic">
33          The query string is '%s'.</p>""" % cgi.escape( query )
34      pairs = cgi.parse_qs( query )
35
36      for key, value in pairs.items():
37          print "<p>You set '%s' to value %s</p>" % \
38              ( key, value )
39
40  print "</body></html>"
```

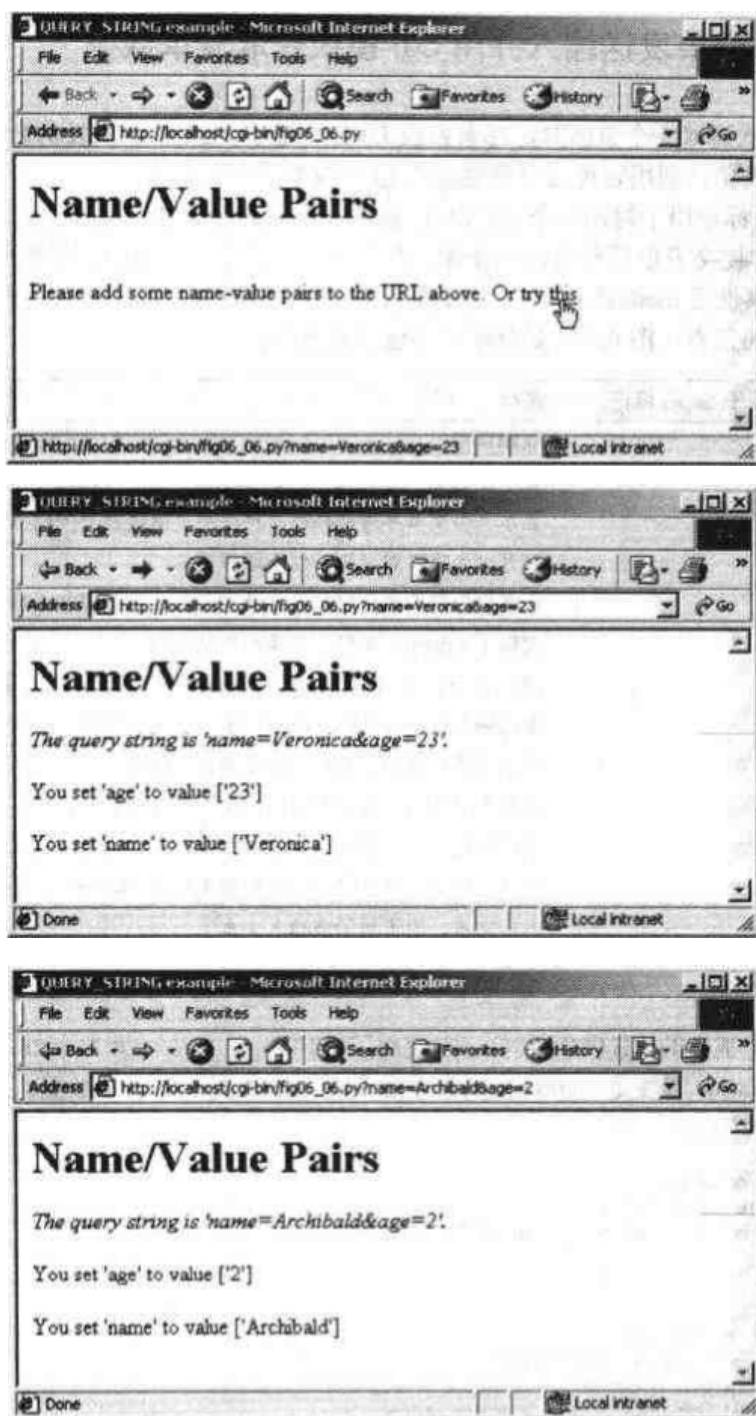


图 6.6 从 QUERY_STRING 读取输入

在图 6.6 的第 23 行, 我们将环境变量 QUERY_STRING 的值指派给变量 query。然后, 第 25 行检测 query 是否为空。如果为空, 将打印一条消息, 指示用户在 URL 中添加查询字符串。还提供了一个 URL 链接, 它包含了示范查询字符串。注意查询字符串数据也可指定为网页的一个超文本链接的一部分。

如果查询字符串不为空, 就会打印查询字符串的值 (第 32~33 行)。函数 cgi.parse_qs 解析 (即分解) 查询字符串 (第 34 行)。该函数要取一个查询字符串作为参数, 并返回一个由其中的名-值对构成的字典。第 36~38 行是一个 for 循环, 它打印字典 pairs 中包含的名和值。

6.5 用 XHTML 表单发送输入并用 cgi 模块获取表单数据

用户不可能在每次访问一个页面时，都亲自在 URL 中输入 CGI 脚本需要的所有信息。XHTML 允许在网页中包含一个表单，利用它可更直观地输入 CGI 脚本需要的信息。

`<form>`和`</form>`标记用于封闭一个 XHTML 表单。`<form>`标记通常要设置两个属性。第一个属性是 `action`，它指定用户提交表单后要执行的操作。在我们的例子中，操作通常是调用一个 CGI 脚本来处理表单数据。第二个属性是 `method`，它可为 `get` 或 `post`。本节要展示使用了这两种方法的例子。XHTML 表单可包括任意数量的元素。图 6.7 简要总结了可包括的元素。

标记名	type 属性 (用于 <input 标记=""/>)	说明
<code><input></code>	<code>button</code>	标准按钮
	<code>checkbox</code>	显示复选框，可勾选 (<code>true</code>) 或清除勾选 (<code>false</code>)
	<code>file</code>	显示一个文本字段和按钮，允许用户指定要上传到 Web 服务器的文件。按钮将打开一个文件对话框，便于选择文件
	<code>hidden</code>	在客户面前隐藏数据信息，使隐藏表单数据只能由服务器的表单处理程序使用
	<code>image</code>	类似于 <code>submit</code> ，但显示图像而不是按钮
	<code>password</code>	类似于 <code>text</code> ，但输入的每个字符都呈现为星号 (*)，以隐藏输入，确保安全
	<code>radio</code>	单选按钮类似于复选框，区别在于一组单选按钮中，一次只能选一个
	<code>reset</code>	显示重置按钮，将表单字段重置为默认值
	<code>submit</code>	显示提交按钮，根据表单的 <code>action</code> 来提交表单数据
	<code>text</code>	提供单行文本字段，以便输入文本。该属性是默认的输入类型
<code><select></code>		显示下拉菜单或选择框。如果用于标记 <code><option></code> ， <code><select></code> 就表明要选择的项目
<code><textarea></code>		多行文本，用于显示或输入文本

图 6.7 XHTML 表单元素

图 6.8 演示了一个基本的 XHTML 表单，它使用了 HTTP `get` 方法。第 21~26 行输出表单。注意，`method` 属性是 `get`，`action` 属性是 `fig06_08.py`（也就是说，一旦提交了表单数据，脚本会调用自己来处理数据，这称为“回传”）。

```

1  #!c:\Python\python.exe
2  # Fig. 6.8: fig06_08.py
3  # Demonstrates get method with an XHTML form.
4
5  import cgi
6
7  def printHeader( title ):
8      print """Content-type: text/html
9
10     <?xml version = "1.0" encoding = "UTF-8"?>
11     <!DOCTYPE html PUBLIC
12         "-//W3C//DTD XHTML 1.0 Strict//EN"
13         "DTD/xhtml1-strict.dtd">
14     <html xmlns = "http://www.w3.org/1999/xhtml">
15     <head><title>%s</title></head>
16
17     <body>""" % title
18
19  printHeader( "Using 'get' with forms" )
20  print """<p>Enter one of your favorite words here:<br /></p>
21     <form method = "get" action = "fig06_08.py">
22         <p>
23             <input type = "text" name = "word" />
24             <input type = "submit" value = "Submit word" />
25         </p>

```

```

26     </form>"""
27
28 pairs = cgi.parse()
29
30 if pairs.has_key( "word" ):
31     print """<p>Your word is:
32         <span style = "font-weight: bold">%s</span></p>""" \
33         % cgi.escape( pairs[ "word" ][ 0 ] )
34
35 print "</body></html>"

```



图 6.8 get 用于 XHTML 表单

表单包含两个输入字段。第一个是单行文本字段 (`type = "text"`)，名称是 `word` (第 23 行)。第二个显示 “Submit word” 按钮，用于提交表单数据 (第 24 行)。

脚本首次执行时，`QUERY_STRING` 不包含任何值 (除非用户有意为 URL 附加了一个查询字符串)。然而，一旦用户在 `word` 文本字段输入一个单词，并单击了 “Submit word” 按钮，脚本就会被再度调用。这一次，`QUERY_STRING` 环境变量包含了文本输入字段的名称 (`word`) 以及用户输入的值。例如，假定用户输入单词 `python`，并单击 “Submit word” 按钮，`QUERY_STRING` 包含的值就会是 “`word=python`”。

第 28 行使用函数 `cgi.parse` 来解析表单数据。该函数类似函数 `cgi.parse_qs`，不过 `cgi.parse` 只解析来自标准输入 (而不是查询字符串) 的数据，并用一个字典返回名-值对。

所以，脚本第二次执行时，一旦查询字符串得以解析，第 28 行就将返回的字典指派给变量 `pairs`。如字典 `pairs` 包含 “`word`” 这个键，表明用户至少提交了一个单词，程序会将单词打印到浏览器。如果输入中含有特殊字符 (比如 `<`、`>` 或空格)，单词就会被传给函数 `cgi.escape`。第 31~33 行用 CSS 来显示结果。在图 6.8 中，可看到地址栏中的空格被加号取代，因为 Web 浏览器的 URL 会对它们发送的 XHTML 表单数据进行编码，即空格转换成加号，其他一些符号 (比如省略号) 则转换成相应的 ASCII 编码 (采用十六进制，前面要添加一个百分号)。

和图 6.6 一样，这里利用环境变量，通过 `get` 与 XHTML 表单的配合使用，将数据传给 CGI 脚本。CGI 脚本与服务器交互的另一种方式是使用标准输入和 `post` 方法。为便于比较，下面要用 `post` 重新实现图 6.8 的应用程序 (参见图 6.9)。注意，两者的代码几乎完全一致。XHTML 表单指出现在要用 `post` 方

法提交表单数据（第 21 行）。

```

1  #!c:\Python\python.exe
2  # Fig. 6.9: fig06_09.py
3  # Demonstrates post method with an XHTML form.
4
5  import cgi
6
7  def printHeader( title ):
8      print """Content-type: text/html
9
10 <?xml version = "1.0" encoding = "UTF-8"?>
11 <!DOCTYPE html PUBLIC
12     "-//W3C//DTD XHTML 1.0 Strict//EN"
13     "DTD/xhtml1-strict.dtd">
14 <html xmlns = "http://www.w3.org/1999/xhtml">
15 <head><title>%s</title></head>
16
17 <body>""" % title
18
19 printHeader( "Using 'post' with forms" )
20 print """<p>Enter one of your favorite words here:<br /></p>
21     <form method = "post" action = "fig06_09.py">
22         <p>
23             <input type = "text" name = "word" />
24             <input type = "submit" value = "Submit word" />
25         </p>
26     </form>"""
27
28 pairs = cgi.parse()
29
30 if pairs.has_key( "word" ):
31     print """<p>Your word is:
32         <span style = "font-weight: bold">%s</span></p>""" \
33         % cgi.escape( pairs[ "word" ][ 0 ] )
34
35 print "</body></html>"

```

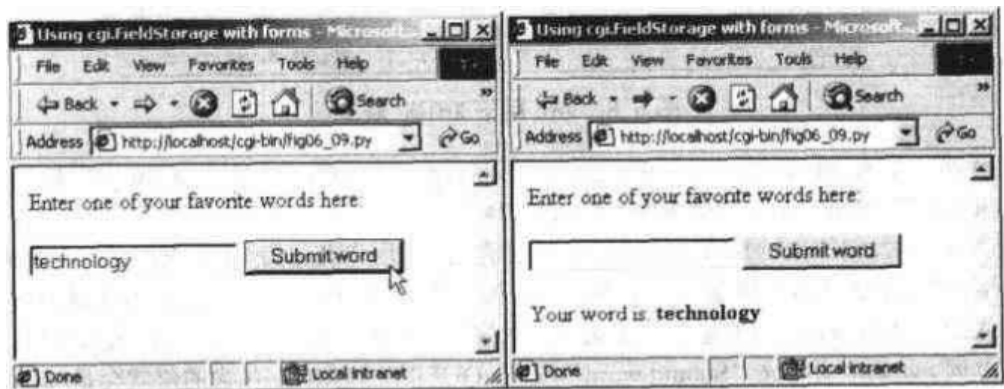


图 6.9 post 与 XHTML 表单配合使用

post 方法通过标准输入将数据发送给 CGI 脚本。数据采用和在 QUERY_STRING 中一样的方式进行编码（即使用等号和&符号连接名-值对），但没有设置 QUERY_STRING 环境变量。相反，post 方法设置了环境变量 CONTENT_LENGTH，指出发送（post）的字符数。post 方法的一个好处就是字符数可变。

尽管 get 和 post 方法非常相似，但仍然存在一些重要区别。get 请求将表单内容作为 URL 的一部分发送。post 请求将表单内容附加到 HTTP 请求的末尾发送。另一个区别是浏览器对响应进行处理的方式。浏览器通常要缓存网页，所以再次请求网页时，浏览器不会再次下载，而是从缓存中载入页。这加快了浏览速度，并减少了查看网页所需的下载量。然而，浏览器不缓存服务器对 post 请求的响应，因为后续的 post 请求包含的信息也许不相同。

这和处理 get 请求的响应是不同的，使用基于 Web 的搜索引擎时，get 请求通常向搜索引擎提供

XHTML 表单中指定的信息。然后, 搜索引擎执行搜索, 并以网页形式返回结果。

软件工程知识 6.2 大多数 Web 服务器将 get 请求查询字符串限定在 1024 个字符以下, 为避免这一限制, 请用 post 请求。

软件工程知识 6.3 包含许多字段的表单通常使用 post 请求。密码等敏感表单字段通常用 post 请求发送。

6.6 用 cgi.FieldStorage 读取输入

图 6.10 重新实现了图 6.9 的例子, 它利用了 cgi 模块所提供的高级数据抽象。第 28 行创建类 cgi.FieldStorage 的一个对象 (类的详情在第 7 章讨论)。在我们的例子中, 高级数据类型 (或类) 叫做 cgi.FieldStorage, 类似于解析函数所返回的字典。

```

1  #!c:\Python\python.exe
2  # Fig. 6.10: fig06_10.py
3  # Demonstrates use of cgi.FieldStorage an with XHTML form.
4
5  import cgi
6
7  def printHeader( title ):
8      print """Content-type: text/html
9
10 <?xml version = "1.0" encoding = "UTF-8"?>
11 <!DOCTYPE html PUBLIC
12     "-//W3C//DTD XHTML 1.0 Strict//EN"
13     "DTD/xhtml11-strict.dtd">
14 <html xmlns = "http://www.w3.org/1999/xhtml">
15 <head><title>%s</title></head>
16
17 <body>""" % title
18
19 printHeader( "Using cgi.FieldStorage with forms" )
20 print """<p>Enter one of your favorite words here:<br /></p>
21     <form method = "post" action = "fig06_10.py">
22         <p>
23             <input type = "text" name = "word" />
24             <input type = "submit" value = "Submit word" />
25         </p>
26     </form>"""
27
28 form = cgi.FieldStorage()
29
30 if form.has_key( "word" ):
31     print """<p>Your word is:
32         <span style = "font-weight: bold">%s</span></p>""" \
33         % cgi.escape( form[ "word" ].value )
34
35 print "</body></html>"

```

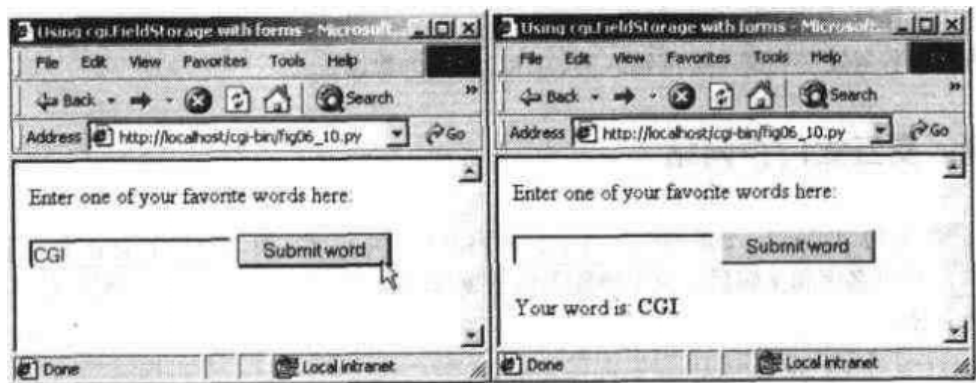


图 6.10 cgi.FieldStorage 与 XHTML 表单配合使用

第 30 行调用字典方法 `has_key` 并传递 `form`，判断字典是否包含“word”键。如答案是肯定的，表明用户输入了一个单词，所以程序将单词打印到浏览器（第 31~33 行）。注意，要想访问 `cgi.FieldStorage` 对象中任何键的值，必须访问键所对应的值的 `value` 属性。

6.7 其他 HTTP 标头

6.2.3 节指出，除标准 HTTP 标头（`Content-type: text/html`）之外，还有其他许多 HTTP 标头。例如：

```
print "Content-type: text/plain"
```

将用 `text/plain` 内容类型打印 `Content-type` 头。如一个页的 `Content-type` 是 `text/plain`，整个页都会被作为纯文本（而非一个 HTML 或 XHTML 文档）进行处理。

除 `Content-type` 之外，CGI 脚本还可提供其他 HTTP 标头。大多数情况下，服务器会原封不动地将这些额外的头传给客户。例如，经过指定时间后，`Refresh` 头将客户重定向至位置：

```
Refresh: "5; URL = http://www.deitel.com/newpage.html"
```

Web 浏览器收到这个头的 5 秒钟之后，会自动请求指定 URL 处的资源。此外，如果 `Refresh` 头省略了 URL 部分，会按指定时间周期来刷新当前页。

CGI 协议规定，由 CGI 脚本输出的特定类型的头要由服务器处理，而不是直接传给客户。第一个是 `Location` 头。类似于 `Refresh` 头，`Location` 将客户重定向到新位置：

```
Location: http://www.deitel.com/newpage.html
```

与相对 URL 结合使用（例如 `Location: /newpage.html`），`Location` 头可向服务器指出“重定向将在服务器端执行”，而不是将 `Location` 头发送回客户。在这种情况下，用户感觉最初是由浏览器请求的资源。Python 脚本如果使用了 `Location` 头，就不再需要 `Content-type` 头，因为新资源有自己的内容类型。

CGI 规范还支持一个 `Status` 头，它要求服务器输出一个状态头行（比如 `HTTP/1.1 200 OK`）。通常，服务器会向客户发送适当的头行（例如，在大多数情况下会添加 200 OK 状态码）。但是，CGI 允许更改响应状态。例如，假如发送以下头信息：

```
Status: 204 No Response
```

表明即使请求成功，浏览器也要继续显示相同的页。如希望用户在提交表单后不转移到一个新页，这个头就非常有用。

前面介绍了 CGI 协议基本知识。总之，CGI 允许脚本通过 3 种基本方式与服务器交互：

1. 通过标准输出向客户输出头和内容。
2. 利用服务器的环境变量设置（包括 URL 编码的 `QUERY_STRING`），这些变量的值可在脚本内使用（通过 `os.environ`）。
3. 通过服务器发送到脚本标准输入的、URL 编码的数据。

6.8 示例：交互式门户网站

图 6.11 和图 6.12 实现了一个简单的交互式门户网站（登录页），它用于虚拟的 Bug2Bug Travel 网站。它们查询客户，获得名字和密码后，再根据输入的数据显示信息。为简化问题，本例没有对发送给服务器的数据进行加密。

```
1 <?xml version = "1.0" encoding = "UTF-8"?>
2 <!DOCTYPE html PUBLIC
```

```

3  "-//W3C//DTD XHTML 1.0 Strict//EN"
4  "DTD/xhtml11-strict.dtd">
5  <!-- Fig. 6.11: fig06_11.html -->
6  <!-- Bug2Bug Travel log-in page. -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9    <head><title>Enter here</title></head>
10
11    <body>
12      <h1>Welcome to Bug2Bug Travel</h1>
13
14      <form method = "post" action = "/cgi-bin/fig06_12.py">
15
16        <p>Please enter your name:<br />
17        <input type = "text" name = "name" /><br />
18
19        Members, please enter the password:<br />
20        <input type = "password" name = "password" /><br />
21        </p>
22
23        <p style = "font-size: em - 1; font-style: italic" >
24        Note that password is not encrypted.<br /><br />
25        <input type = "submit" />
26        </p>
27
28      </form>
29    </body>
30  </html>

```



图 6.11 交互式门户网站创建了一个用密码保护的网页

```

1  #!c:\Python\python.exe
2  # Fig. 6.12: fig06_12.py
3  # Handles entry to Bug2Bug Travel.
4
5  import cgi
6
7  def printHeader( title ):
8    print """Content-type: text/html
9
10 <?xml version = "1.0" encoding = "UTF-8"?>
11 <!DOCTYPE html PUBLIC
12   "-//W3C//DTD XHTML 1.0 Strict//EN"
13   "DTD/xhtml11-strict.dtd">
14 <html xmlns = "http://www.w3.org/1999/xhtml">
15 <head><title>%s</title></head>
16
17 <body>""" % title
18

```

```

19 form = cgi.FieldStorage()
20
21 if not form.has_key( "name" ):
22     print "Location: /fig06_11.html\n"
23 else:
24     printHeader( "Bug2Bug Travel" )
25     print "<h1>Welcome, %s!</h1>" % form[ "name" ].value
26     print "<p>Here are our weekly specials:<br /></p>"
27     <ul><li>Boston to Taiwan for $300</li></ul>""
28
29 if not form.has_key( "password" ):
30     print "<p style = 'font-style: italic'"
31         Become a member today for more great deals!</p>""
32 elif form[ "password" ].value == "Coast2Coast":
33     print "<p><br />"
34     <p>Current specials just for members:<br /></p>"
35     <ul><li>San Diego to Hong Kong for $250</li></ul>""
36 else:
37     print "<p style = 'font-style: italic'"
38         Sorry, you have entered the wrong password.
39         If you have the correct password, enter
40         it to see more specials.</p>""
41
42 print "<hr /></body></html>"

```

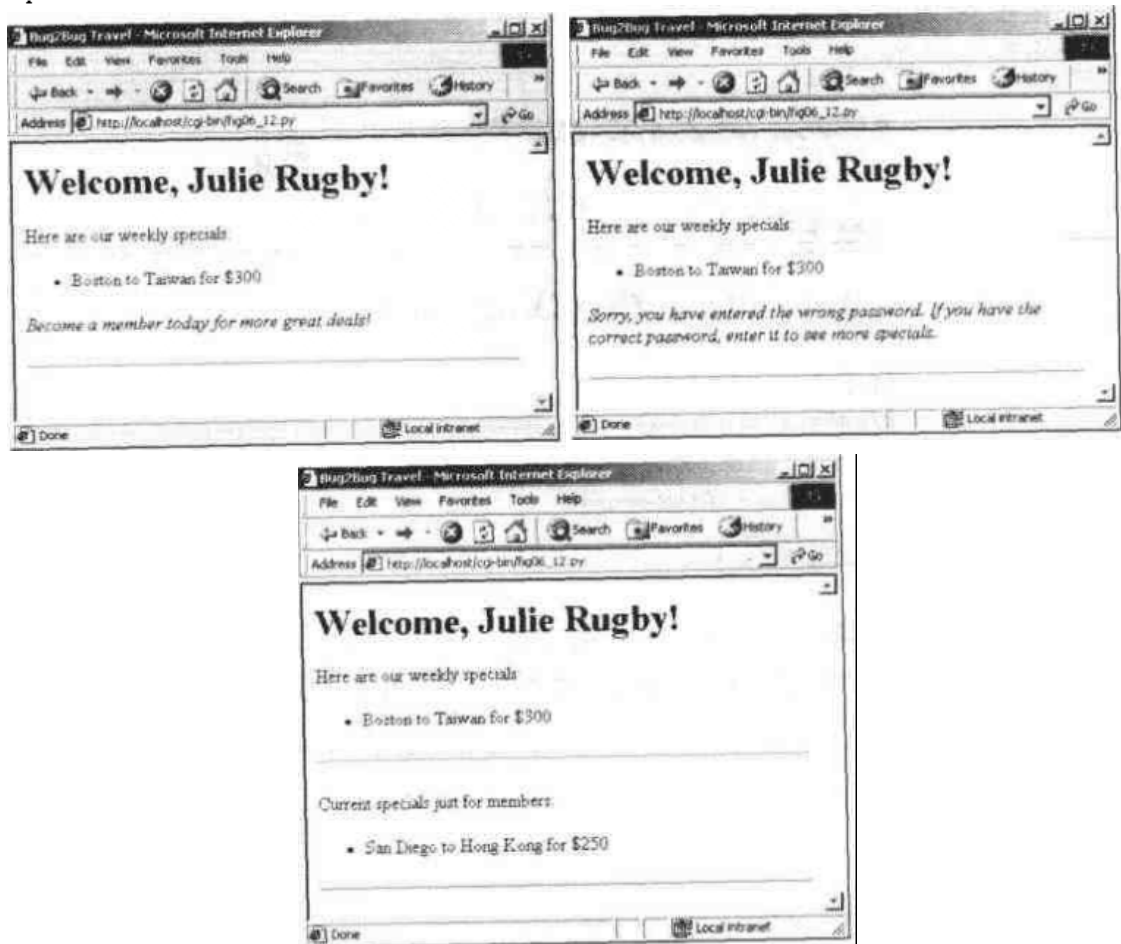


图 6.12 交互式门户网站处理程序

图 6.11 显示了起始页。它是一个静态 XHTML 文档，包含一个表单，可将数据 post 给 CGI 脚本 fig06_12.py（第 14 行）。表单用一个字段收集客户的名字（第 17 行），用一个字段收集会员密码（第 20 行）。为了能从 Apache 服务器使用该 XHTML 文件，要将 fig06_11.html 放到 Apache 服务器根目录（例如 Apache Group\Apache\htdocs）。欲了解 Apache 服务器的详情，请访问 www.apache.org。

图 6.12 是处理客户数据的 CGI 脚本。第 20 行从一个 `cgi.FieldStorage` 实例获得了表单数据，并将结果指派给局部变量 `form`。开始于第 22 行的 `if` 结构检测 `form` 里是否包含名为“name”的键。如果 `form` 不包含那个键，就表明用户尚未输入一个名字，所以打印一个 Location HTTP 标头（第 23 行），将用户重定向到可以输入名字的 XHTML 文件（`fig06_11.html`）。文档 `fig06_11.html` 包含在 Web 服务器的主文档根中（因为文件名前有一个正斜杠/）。第 23 行的效果是，客户如果想不经过登录过程就直接访问 `fig06_12.py`，那就必须经由门户进入。

如果用户输入了名字，就打印欢迎消息，包括用户的名字以及每周特价（第 26~28 行）。第 30 行检测用户是否输入密码。如果没有输入密码，就邀请用户升级为会员（第 31 行）。如果输入了密码，第 32 行就判断密码是否等于字符串“Coast2Coast”。如果是，就在浏览器中打印会员特价。注意，本例的密码、每周特价和会员特价都是“硬编码”的（也就是说，它们的值直接在代码中提供）。如果用户输入的密码不等于“Coast2Coast”，应用程序就会要求用户输入一个有效的密码（第 36~38 行）。

性能提示 6.1 为响应每个 CGI 请求，Web 服务器要执行 CGI 程序以便为客户创建响应。这个过程要比返回静态文档花费更多时间。实现一个网站时，将变化不频繁的内容定义成静态内容。和只使用 CGI 脚本相比，这可使 Web 服务器更快地响应。

6.9 因特网和万维网资源

www.w3.org/CGI

W3C 的 CGI 主页，其中的 CGI 安全问题值得一读。此外，这里还提供了到 CGI 规范的链接。注意 CGI 规范最早是由 NCSA（美国国家超级计算机应用中心）制定的。

www.nacs.uci.edu/indiv/ehood/MIME/MIME.html

这里提供了到 MIME RFC（注释请求）、MIME 相关 RFC 以及与 MIME 相关的其他信息的链接。

www.speakeasy.org/~cgires

这里收集了大量 CGI 教程和脚本。

www.fastcgi.com

“快速 CGI”的主页，快速 CGI 是对 CGI 的一个扩展，面向高性能的 Internet 应用。

bel-epa.com/pyapache

PyApache 的资源中心。PyApache 是一个模块，用于将 Python 嵌入 Apache 服务器。

www.modpython.org

`mod_python` 主页。`mod_python` 也是一个模块，用于将 Python 解释器嵌入 Apache 服务器，使脚本运行速度超过传统的 CGI 脚本。

第 7 章 基于对象的编程

学习目标

- 理解“封装”和“数据隐藏”的软件工程概念
- 理解数据抽象和抽象数据类型（ADT）记号法
- 会创建 Python ADT（即类）
- 理解如何创建、使用 and 销毁对象
- 控制对象属性和方法访问
- 体会面向对象的重要价值

7.1 概述

现在开始较深入地学习面向对象。第 2~6 章讨论 Python 程序时，已经提到了许多基本概念（即“对象思考”）和术语（即“对象语言”）。这里简要总结一下面向对象的关键概念和术语。面向对象编程（OOP）将数据（属性）和函数（行为）封装到名为“类”的组件中。一个类的数据和函数紧密联系在一起。类好比蓝图。根据蓝图，建筑师可建造一座房屋。同样地，根据类，程序员可创建一个对象（也称为实例）。蓝图可多次重用，从而建造许多房屋；类也可多次重用，创建同一个类的多个对象。类具有“信息隐藏”的特点。这意味着尽管对象知道如何通过良好定义的“接口”相互通信，但通常不允许一个对象知道另一个类是如何实现的——实现细节隐藏在类中。我们完全能很好地驾驶一辆车，根本不必去了解发动机、传动机构和排气系统的内部工作原理。后文将解释信息隐藏对于保障“良好软件工程”的重要性。

在 C 和其他过程式程序语言中，编程通常是“面向行动”的；在 Python 中，编程则可以“面向对象”。在过程式语言中，基本编程单元是“函数”；在面向对象语言中，基本编程单元是“类”，最终要通过它来实例化（即“创建”）对象。

采用过程式编程，程序员把重点放在写函数上。把用于执行一些任务的行动组合成函数，不同的函数进一步组合，即构成程序。在过程式编程中，数据肯定是重要的，但数据的主要用途是为函数执行的行动提供支持。在一份系统规格说明中（描述应用程序所提供的服务的一份文档），那些动词帮助过程式程序员确定并设计一系列协同工作的函数，它们用于实现整个系统。

采用面向对象编程，程序员则把重点放在创建他们自己的用户自定义类型上，即“类”。类也称为“程序员自定义类型”。每个类都包含数据和一系列数据处理函数。类的数据组件称为属性或数据成员；类的函数组件称为方法（在其他面向对象编程语言中，也可能称为“成员函数”）。面向对象编程把重点集中于类，而不是函数。在一份系统规格说明中，那些名词帮助面向对象程序员确定并设计一系列类，以便通过它们创建一系列协同工作的对象，这些对象用于实现整个系统。

软件工程知识 7.1 本书中心思想是“重用，重用，还是重用”。我们将详细讨论用于精心编写类的技巧，以促进重用。我们的重点是“创建宝贵的类”，创造有价值的“软件资产”。

7.2 用类实现一个 Time 抽象数据类型

用类可为对象建模，对象有自己的数据（表示成“属性”）和行为（表示成“方法”）。为响应发送给对象的消息，需要调用方法。一条消息对应于从一个对象发送给另一个对象的一个方法调用。

类简化了编程，因为客户（类的用户）只需关心对象中封装或嵌入的操作——即“对象接口”。这些操作通常设计成面向客户，而非面向实现。客户无需关心类的具体实现（当然，客户希望的是正确而且高效的实现）。一旦实现发生改变，依赖于实现的代码必须相应改变。通过隐藏实现，有助于避免其他

程序部件依赖于类的实现细节。

类通常不必从头创建。相反，它们可从其他类派生，后者提供了新类可利用的属性和行为。另外，一个类也可将其他类的对象作为自己的成员。这样的“软件重用”能极大提高编程效率。从现有的类派生出新类称为“继承”，这将在第9章详细讲解。

图 7.1 包含 Time 类的一个简单定义。该类包含对一天中的时间进行描述的信息，并包含以两种格式打印时间的方法。类内部以 24 小时格式维护时间（军事时间），但允许客户采用 24 小时格式或标准（AM, PM）格式显示时间。本节后面展示了一个程序（图 7.2），它演示了如何创建 Time 类的一个对象。

关键字 class（第 4 行）开始一个类定义。该关键字后面是类名（Time），再后面是一个冒号（:）。包含关键字 class 和类名的这一行称为“类头”。类的“主体”是一个缩进的代码块（第 5~37 行），其中包含从属于该类的方法和属性。类通常遵循与变量一样的命名约定，不过类名的首字母必须大写。

```

1 # Fig. 7.1: Time1.py
2 # Simple definition of class Time.
3
4 class Time:
5     """Time abstract data type (ADT) definition"""
6
7     def __init__( self ):
8         """initializes hour, minute and second to zero"""
9
10        self.hour = 0      # 0-23
11        self.minute = 0    # 0-59
12        self.second = 0    # 0-59
13
14    def printMilitary( self ):
15        """Prints object of class Time in military format"""
16
17        print "%.2d:%.2d:%.2d" % \
18            ( self.hour, self.minute, self.second ),
19
20    def printStandard( self ):
21        """Prints object of class Time in standard format"""
22
23        standardTime = ""
24
25        if self.hour == 0 or self.hour == 12:
26            standardTime += "12:"
27        else:
28            standardTime += "%d:" % ( self.hour % 12 )
29
30        standardTime += "%.2d:%.2d" % ( self.minute, self.second )
31
32        if self.hour < 12:
33            standardTime += " AM"
34        else:
35            standardTime += " PM"
36
37        print standardTime,
```

图 7.1 Time 类包含用于存储和显示时间的属性及方法

常见编程错误 7.1 忘记在类定义头的末尾添加冒号是语法错误。

常见编程错误 7.2 类的主体没有缩进是语法错误。

第 5 行是类的可选的“文档化字符串”，它描述了类。要想包含文档化字符串，该字符串必须位于紧接在类头之后的一行或多行中。要查看类的文档化字符串，应执行以下语句：

```
print ClassName.__doc__
```

模块、方法和函数也可指定一个文档化字符串。

良好编程习惯 7.1 尽可能包含文档化字符串，使程序更有条理。

良好编程习惯 7.2 文档化字符串习惯上是一个三引号字符串。这样可在不改变引号样式的前提下，扩展一个程序的文档（例如添加更多的行）。

第 7 行开始定义特殊方法 `__init__`，它是类的“构造函数”方法。每次创建类的一个对象时，都会执行它的构造函数。构造函数（`__init__` 方法）会初始化对象属性，并返回 `None`。Python 类还可定义另外几个特殊方法，这些方法之前和之后都有双下划线（`__`）。详情将在第 8 章讨论。

常见编程错误 7.3 如果从构造函数返回的值不是 `None`，就属于严重的运行时错误。

软件工程知识 7.2 先初始化对象，再让客户代码调用对象的方法。不能依赖客户代码正确初始化对象。

良好编程习惯 7.3 如有必要（几乎总是如此），请提供一个构造函数，用有意义的值初始化每个对象。

包括构造函数在内的所有方法至少要指定一个参数。该参数代表要调用其方法的类的对象。人们经常把这个参数称为“类实例对象”。但由于该术语容易混淆，所以我们将任何方法的第一个参数都称为“对象引用参数”，或简称“对象引用”。方法必须通过对象引用来访问从属于类的属性以及其他方法。按照约定，对象引用参数称为 `self`。

常见编程错误 7.4 忘记将对象引用（通常是 `self` 参数）设为方法定义中的第一个参数，会导致该方法在运行时被调用时，造成严重逻辑错误。

良好编程习惯 7.4 将所有方法的第一个参数都命名为 `self`。始终遵循这一命名约定，可确保不同程序员编写的 Python 程序是一致的。

每个对象都有自己的命名空间，其中包含对象的方法和属性。类的构造函数用一个空对象（`self`）开头，并将属性添加到对象的命名空间。例如，`Time` 类的构造函数（第 7~12 行）将 3 个属性（`hour`，`minute` 和 `second`）添加到新对象的命名空间。第 10 行将属性 `hour` 同对象的命名空间绑定，并将属性值初始化为 0。一旦属性添加到对象的命名空间，使用对象的客户就能访问属性值。

`Time` 类还定义了方法 `printMilitary` 和 `printStandard`。注意，方法也能指定一个文档化字符串，只是同样要紧接在方法头之后的一行或多行中。本例中，每个方法都指定一个参数（`self`），它引用要调用其方法的类的对象。每个方法都通过参数 `self` 来访问对象属性。本例中，`printMilitary` 方法（第 14~18 行）以军事（24 小时）格式打印时间；`printStandard` 方法（第 20~37 行）则以标准（12 小时）格式打印时间。

一旦定义好类，程序就能创建那个类的对象。可存在一个类的许多对象，程序员可根据需要创建它们。正是由于这个原因，我们认为 Python 是一种“可扩展语言”。图 7.2 的程序创建 `Time` 类的一个对象。首先从 `Time1.py` 导入类定义。注意，第 4 行导入定义的方式和程序从模块导入任何元素的方式是一样的。

```
1 # Fig. 7.2: fig07_02.py
2 # Creating and manipulating objects of class Time.
3
4 from Time1 import Time # import class definition from file
5
6 time1 = Time() # create object of class Time
7
8 # access object's attributes
9 print "The attributes of time1 are: "
10 print "time1.hour:", time1.hour
11 print "time1.minute:", time1.minute
12 print "time1.second:", time1.second
13
14 # access object's methods
15 print "\nCalling method printMilitary:",
16 time1.printMilitary()
17
```

```

18 print "\nCalling method printStandard:",
19 timel.printStandard()
20
21 # change value of object's attributes
22 print "\n\nChanging timel's hour attribute..."
23 timel.hour = 25
24 print "Calling method printMilitary after alteration:",
25 timel.printMilitary()

```

```

The attributes of timel are:
timel.hour: 0
timel.minute: 0
timel.second: 0

Calling method printMilitary: 00:00:00
Calling method printStandard: 12:00:00 AM

Changing timel's hour attribute...
Calling method printMilitary after alteration: 25:00:00

```

图 7.2 创建对象

良好软件工程的一项基本原则在于，客户不必知道类的实现细节，就能使用那个类。由于 Python 使用了模块，所以极大简化了这种数据抽象。图 7.2 的程序直接导入 Time 定义，并在不知道类的实现细节这一前提下，使用 Time 类。

软件工程知识 7.3 类的客户不必访问类的源代码，就可使用这个类。

要创建 Time 类的对象，只需调用类名，如同它是一个简单的函数（第 6 行）。这实际调用的是 Time 类的构造函数。尽管类定义规定构造函数（即 `__init__`）要取得一个参数，但第 6 行没有向其传递任何参数。Python 将第一个（对象引用）参数插入每个方法调用，其中也包括类的构造函数调用。构造函数初始化对象的属性。一旦构造函数退出，Python 就将新建的对象指派给 `timel`。

客户代码必须通过一个对象引用来访问对象的属性。第 10~12 行演示了程序怎样通过点访问运算符（`.`）来访问一个对象的属性。对象名在点的左侧，属性在点的右侧。输出结果演示了由构造函数指派给属性 `hour`、`minute` 和 `second` 的初始值。

客户代码可采取类似方式访问对象的方法。第 16 行调用 `timel` 的 `printMilitary` 方法。注意，方法调用同样没有传递参数，尽管方法定义规定了一个名为 `self` 的参数。Python 在 `printMilitary` 调用中传递对 `timel` 的一个引用，使方法能访问对象的属性。

第 23 行修改了指派给 `timel.hour` 属性的值。第 24~25 行的输出展示了客户随意访问对象数据时极易发生的一个问题。`hour` 属性的含义变得不明确，因为数据成员现在的值是 25。我们说数据成员处于“不一致状态”（包含无效值）。其他程序语言提供了相应的机制来防止客户访问一个对象的数据。但另一方面，Python 没有提供这种严格的编程结构。本章后面将讨论 Python 程序员用于确保对象数据处于一致状态的各种方法。

常见编程错误 7.5 直接访问对象的属性可能导致数据进入不一致状态。

7.3 特殊属性

类和类的对象都具有可供操纵的特殊属性。定义类或创建类的一个对象时，Python 会创建这些属性，它们提供了与类或类的对象有关的信息。图 7.3 总结了所有类都包含的特殊属性。图 7.4 的交互式会话则打印 Time 类的所有这些属性。

此外，类的所有对象也具有一些公共属性。图 7.5 对它们进行了小结，图 7.6 的交互式会话打印了 Time 类的一个对象的属性值。注意，对象可访问从属于类的 `__doc__` 及 `__module__` 属性。

这些属性使 Python 具有强大的“自省”能力（即提供有关自己的信息的能力）。许多程序员利用这

些能力创建高级的、动态和灵活的应用程序。本书主要利用这些能力探索 Python 的工作方式，以便读者可进一步理解编程概念。

属性	说明
<code>__bases__</code>	包含基类的一个元组，类可从这些基类直接继承。如果类不从其他类继承，元组就会为空。第 9 章会详细地讨论基类和继承
<code>__dict__</code>	与类的命名空间对应的一个字典。其中每个键-值对都代表在命名空间中的一个标识符及其值
<code>__doc__</code>	类的文档化字符串。如果类没有指定文档化字符串，值为 None
<code>__module__</code>	包含模块（文件）名的一个字符串，类定义于这个模块中
<code>__name__</code>	包含类名的一个字符串

图 7.3 类的特殊属性

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from Timel import Time
>>> print Time.__bases__
()
>>> print Time.__dict__
{'printMilitary': <function printMilitary at 0x0079BF80>, '__module__': 'Timel', '__doc__':
'Time abstract data type (ADT) definition', '__init__': <function __init__ at 0x0077AB00>,
'printStandard': <function printStandard at 0x00769990>}
>>>
>>> print Time.__doc__
Time abstract data type (ADT) definition
>>> print Time.__module__
Timel
>>> print Time.__name__
Time
```

图 7.4 类的特殊属性

属性	说明
<code>__class__</code>	对类的一个引用，对象是从该类实例化而来的
<code>__dict__</code>	与对象的命名空间对应的一个字典。其中每个键-值对都代表在命名空间中的一个标识符及其值

图 7.5 类对象的特殊属性

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from Timel import Time
>>> timel = Time()
>>> print timel.__class__
Timel.Time
>>> print timel.__dict__
{'second': 0, 'minute': 0, 'hour': 0}
>>> print timel.__doc__
Time abstract data type (ADT) definition
>>> print timel.__module__
Timel
```

图 7.6 类对象的特殊属性

7.4 控制属性访问

前面讨论了客户如何直接访问对象的属性，并指出这样的操作可能使对象的数据处于不一致状态。大多数面向对象的程序语言都允许禁止直接访问对象的数据。但在 Python 中，程序员要利用属性命名约

定，以便在客户面前隐藏数据。本节将讨论它的优缺点。

7.4.1 get 和 set 方法

尽管客户可直接访问对象的数据（并可能导致数据进入不一致状态），但程序员在设计类时，便可倡导正确的用法。一个办法是让类提供“访问方法”，通过一种得到精心控制的方式来读写类数据。

“断言方法”是一种只读访问方法，用于检测一个条件的有效性。断言方法的一个例子是容器类（能容纳许多对象的一个类）的 `isEmpty` 方法。程序可先调用 `isEmpty`，再从容器对象读取另一个项目。`isFull` 断言方法则检测一个容器对象，判断是否还有额外的空间可用于放置一个数据项。对我们的 `Time` 类来说，合适的断言方法可能是 `isAM` 和 `isPM`。

如果类定义了访问方法，客户就只应通过那些访问方法来访问一个对象的属性。一个典型的操作是通过 `computeInterest` 方法来调整客户账户上的余额（可能是 `BankAccount` 类的一个对象的属性）。

类通常提供了允许客户 `set`（写）或 `get`（读）属性值的方法。尽管这些方法不一定名叫 `set` 和 `get`，但按照约定，它们通常如此。更特别的是，用于设置数据成员 `interestRate` 的方法通常命名为 `setInterestRate`，用于获取 `interestRate` 的方法则通常命名为 `getInterestRate`。`get` 方法也称为“查询”方法。

与直接访问属性相比，`set` 和 `get` 似乎没有多大优势，但要注意两者的一处细微区别。`get` 方法表面上允许客户任意读取数据，但它能控制数据的格式化。而 `set` 方法能够（而且更可能是应该）检查修改属性值的试图。这可保证新值对于数据项来说是恰当的。例如，`set` 方法可拒绝这些不合法的值：日期设为 37，体重为负数，考试成绩为 185 分（如分数范围在 0~100 之间）。

软件工程知识 7.4 利用访问方法控制对属性的访问（尤其是写访问）有助于确保数据完整性。

测试和调试提示 7.1 即使提供了访问方法，也无法自动确保数据完整性，程序员必须提供有效性验证。

类的 `set` 方法有时能返回一些值，表明有人试图为类的对象指派无效数据。这样，类的客户可检测 `set` 方法的返回值，判断处理的是不是一个有效对象；如果是无效对象，则采取相应的行动。另外，`set` 方法可指定一旦发现客户试图为属性指派无效的值，就将一条错误消息（称为“异常”）发送给客户。异常问题将在第 12 章详细讲解。在 Python 中处理无效属性值时，异常是首选的技术。

良好编程习惯 7.5 用于设置数据值的方法应对验证新值的有效性。如果不正确，`set` 方法应报告出错。

软件工程知识 7.5 通过 `set` 和 `get` 方法访问数据，不仅能避免无效数据值，还能将类的客户与数据的表示分开。如果数据表示需要更改（通常是为了减少所需的存储量或者提高性能），只需更改方法主体——只要方法提供的接口保持不变，就不必更改客户。

图 7.7 展示了 `Time2.py`，它定义了修改过的 `Time` 类，该类使用访问方法来保护存储在类中的数据。

```
1 # Fig: 7.7: Time2.py
2 # Class Time with accessor methods.
3
4 class Time:
5     """Class Time with accessor methods"""
6
7     def __init__( self ):
8         """Time constructor initializes each data member to zero"""
9
10        self._hour = 0      # 0-23
11        self._minute = 0    # 0-59
12        self._second = 0    # 0-59
13
14        def setTime( self, hour, minute, second ):
15            """Set values of hour, minute, and second"""
16
17            self.setHour( hour )
```

```
18     self.setMinute( minute )
19     self.setSecond( second )
20
21 def setHour( self, hour ):
22     """Set hour value"""
23
24     if 0 <= hour < 24:
25         self._hour = hour
26     else:
27         raise ValueError, "Invalid hour value: %d" % hour
28
29 def setMinute( self, minute ):
30     """Set minute value"""
31
32     if 0 <= minute < 60:
33         self._minute = minute
34     else:
35         raise ValueError, "Invalid minute value: %d" % minute
36
37 def setSecond( self, second ):
38     """Set second value"""
39
40     if 0 <= second < 60:
41         self._second = second
42     else:
43         raise ValueError, "Invalid second value: %d" % second
44
45 def getHour( self ):
46     """Get hour value"""
47
48     return self._hour
49
50 def getMinute( self ):
51     """Get minute value"""
52
53     return self._minute
54
55 def getSecond( self ):
56     """Get second value"""
57
58     return self._second
59
60 def printMilitary( self ):
61     """Prints Time object in military format"""
62
63     print "%.2d:%.2d:%.2d" % \
64         ( self._hour, self._minute, self._second ),
65
66 def printStandard( self ):
67     """Prints Time object in standard format"""
68
69     standardTime = ""
70
71     if self._hour == 0 or self._hour == 12:
72         standardTime += "12:"
73     else:
74         standardTime += "%d:" % ( self._hour % 12 )
75
76     standardTime += "%.2d:%.2d" % ( self._minute, self._second )
77
78     if self._hour < 12:
79         standardTime += " AM"
80     else:
81         standardTime += " PM"
82
83     print standardTime,
```

图 7.7 为 Time 类定义的访问方法

注意,在第10~12行,构造函数用单一的前置下划线(_)来创建属性。属性名以单下划线开头,虽然在Python语法中没有特殊含义,但单下划线是Python程序员使用类时约定使用的符号,表明程序员不希望类的用户直接访问属性。如果程序要求访问属性,类的作者会提供其他一些途径。本例要求客户通过访问方法来操纵数据。

良好编程习惯 7.6 以单下划线开头的属性揭示一个类的接口的相关信息。类如果定义了此类属性,它的客户就只能通过类提供的访问方法来访问并修改属性值。如果不这样做,通常会导致程序执行期间出现不可预料的错误。

软件工程知识 7.6 Python的类和模块化机制有利于程序的独立实现。如果代码所用的一个类的实现发生了改变,这段代码是无需更改的。

setTime(第14~19行)是set方法,客户要用它设置一个对象的时间中的所有值。该方法要取得属性_hour、_minute和_second的参数值。setHour(第21~27行)、setMinute(第29~35行)以及setSecond(第37~43行)都是针对单独属性的set方法。这些方法为修改时间的客户提供了更大的灵活性。

软件工程知识 7.7 并不是所有方法都要作为类的接口的一部分。有的方法是类的其他方法的一种实用方法,不准备供类的客户使用。

常见编程错误 7.6 如果忘记在方法内部通过对象引用(通常称为self)来访问由对象的类定义的另一个方法,就会导致严重的运行时错误或者逻辑错误。如全局命名空间包含的一个函数与类的某个方法同名,就会产生逻辑错误。此时,如果忘记通过对象引用来访问方法名,实际会调用全局函数。

第24行、第32行和第40行的比较表达式演示了Python的“链式”比较语法,它允许程序员采用熟悉的数学条件来编写比较表达式。使用恰当的and或or操作,也可采用其他语言的语法来改写链式比较表达式。例如,第24行的语句也可写成:

```
hour >= 0 and hour < 24
```

性能提示 7.1 链式比较表达式的效率比非链式表达式更高,因为链式比较表达式中的每个条件都只执行一次。

setHour方法(第21~27行)改变对象的_hour属性。方法检查作为参数传递的值是否在0~23之间(包括0和23)。如小时数有效,方法用新值更新_hour属性。否则,方法会引发一个异常,指出客户试图将对象的数据置入一种不一致的状态。“异常”是一种Python对象,它表明发生了一个特殊事件(通常是错误)。例如,程序试图访问一个不存在的字典键时,Python就会引发一个异常。

第27行的语句使用关键字raise引发一个异常。raise之后是异常名称,一个逗号,再加上异常对象作为属性存储的一个值。Python执行raise语句时,会引发一个异常;如异常未被捕捉,Python会打印一条错误消息,其中包含异常名称以及异常的属性值,如图7.8所示。

```
Python 2.2b2 (#26, Nov. 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> raise ValueError, "This is an error message"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: This is an error message
```

图 7.8 引发异常

setMinute(第29~35行)和setSecond(第37~43行)方法分别更改了属性_minute和_second。每个方法都确保值的范围在0~59之间(包括0和59)。如果值无效,方法会引发异常,并指定恰当的错误和消息参数。

第 45~48 行包含 Time 类的 get 方法。客户用这些方法 (getHour, getMinute 和 getSecond) 分别获取 _hour、_minute 和 _second 属性值。类定义剩余的部分和前面展示的定义是相同的。

软件工程知识 7.8 如果类为其数据提供了访问方法, 客户就只能使用访问方法来获取和修改数据。类和客户的这种“约定”有助于维持数据的一致性状态。

软件工程知识 7.9 类的设计者不需要为每个数据项一一提供 set 或 get 方法; 只有在绝对必要的前提下, 提供这些功能。如果服务对客户来说是合适的, 就应在类的接口中提供这个服务。

软件工程知识 7.10 用于修改对象数据的每个方法都应保证数据处于一致性状态。

图 7.9 是用于修改过的 Time 类的一个 driver。driver 是一个特殊程序, 用于检测类的接口。第 4~6 行从 Time2 模块导入 Time 类, 并创建类的一个对象。第 9~12 行调用 printMilitary 和 printStandard 方法, 显示对象的初始值。

```
1 # Fig. 7.9: fig07_09.py
2 # Driver to test class TimeControl.
3
4 from Time2 import Time
5
6 time1 = Time()
7
8 # print initial time
9 print "The initial military time is",
10 time1.printMilitary()
11 print "\nThe initial standard time is",
12 time1.printStandard()
13
14 # change time
15 time1.setTime( 13, 27, 6 )
16 print "\n\nMilitary time after setTime is",
17 time1.printMilitary()
18 print "\nStandard time after setTime is",
19 time1.printStandard()
20
21 time1.setHour( 4 )
22 time1.setMinute( 3 )
23 time1.setSecond( 34 )
24 print "\n\nMilitary time after setHour, setMinute, setSecond is",
25 time1.printMilitary()
26 print "\nStandard time after setHour, setMinute, setSecond is",
27 time1.printStandard()
```

```
The initial military time is 00:00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

Military time after setHour, setMinute, setSecond is 04:03:34
Standard time after setHour, setMinute, setSecond is 4:03:34 AM
```

图 7.9 调用访问方法来修改数据

第 15 行调用 time1 的 setTime 方法, 传递与 1:27:06 PM 对应的值, 以修改对象的时间值。第 16~19 行调用恰当的方法来显示格式化好的时间。图 7.10 的交互式会话创建 Time 类的一个对象, 并调用 setTime 方法, 试图将对象的数据变成不一致状态。对 setTime 方法的每个调用都包含一个无效值, 每个调用都会造成一条错误消息。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from Time2 import Time
```

```

>>> time1 = Time()
>>>
>>> time1.setHour( 30 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "Time2.py", line 27, in setHour
    raise ValueError, "Invalid hour value: %d" % hour
ValueError: Invalid hour value: 30
>>>
>>> time1.setMinute( 99 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "Time2.py", line 35, in setMinute
    raise ValueError, "Invalid minute value: %d" % minute
ValueError: Invalid minute value: 99
>>>
>>> time1.setSecond( -99 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "Time2.py", line 43, in setSecond
    raise ValueError, "Invalid second value: %d" % second
ValueError: Invalid second value: -99

```

图 7.10 用无效值调用 set 方法

7.4.2 私有属性

在 C++ 和 Java 等程序语言中, 类可明确指出类的客户能访问哪些属性或方法。这些属性或方法被认为是“公共”的。不能由类的客户访问的属性和方法则被认为是“私有”的。

在 Python 中, 对象的属性是肯定能访问的——没有办法阻止其他代码访问数据。然而, Python 提供了一种特别的机制来防止任意访问数据。假定要创建 Time 类的一个对象, 并希望阻止以下赋值语句:

```
time1.hour = 25
```

就可考虑为属性名附加双下划线前缀 (__)。Python 遇到以双下划线开头的属性名时, 解释器会对属性执行“名称重整”(Name Mangling) 操作, 禁止对数据的随意访问。这个操作会改变属性名, 包括与属性所属的类有关的信息。例如, 假如 Time 构造函数包含下面这一行:

```
self.__hour = 0
```

Python 会创建名为 `_Time__hour` 的一个属性, 而不是名为 `__hour` 的一个属性。图 7.11 展示了一个例子, 我们定义类 `PrivateClass`, 其中包含一个公共属性 `publicData` (第 10 行) 和一个私有属性 `__privateData` (第 11 行)。随后的交互式会话 (图 7.12) 演示如何访问对象的数据。

```

1 # Fig. 7.11: Private.py
2 # Class with private data members.
3
4 class PrivateClass:
5     """Class that contains public and private data"""
6
7     def __init__( self ):
8         """Private class, contains public and private data members"""
9
10        self.publicData = "public"      # public data member
11        self.__privateData = "private" # private data member

```

图 7.11 含有私有数据的 PrivateClass 类

```

Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from Private import PrivateClass

```

```

>>> private = PrivateClass()
>>> print private.publicData
public
>>> print private.__privateData
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: PrivateClass instance has no attribute '__privateData'
>>>
>>> print private._PrivateClass__privateData
private
>>> private._PrivateClass__privateData = "modified"
>>> print private._PrivateClass__privateData
modified

```

图 7.12 访问私有数据

在图 7.12 中，首先从模块 `Private` 导入类，再创建名为 `private` 的一个对象。以下语句：

```
print private.publicData
```

会像预期的那样执行——Python 将打印出公共属性的值。但执行以下语句时：

```
print private.__privateData
```

Python 会打印一条错误消息，指出 `PrivateClass` 类不包含名为 `__privateData` 的一个属性。我们为属性名附加了双下划线前缀，所以 Python 会在类定义中更改属性名。

但是仍然可以访问数据，因为 Python 会将 `__privateData` 属性重命名为 `_PrivateClass__privateData`。所以下面这一行：

```
print private._PrivateClass__privateData
```

能成功打印指派给私有属性的值。会话中的最后两个语句演示出，私有数据可采取与公共数据一样的方式进行修改。

但是，如果以这种方式访问和修改私有属性，会违背类作者的数据封装意图。客户永远都不应执行这样的操作，而应坚持使用类提供的访问方法。

软件工程知识 7.11 将客户不应访问的任何数据设为私有。

Python 程序员出于不同原因而使用私有属性。有的程序员通过私有属性来避免继承结构中经常出现的作用域问题（注意，继承问题将在第 9 章讨论）。另一些程序员为禁止客户访问的数据或方法而使用私有属性。这些属性和方法是类的内部工作所必不可少的，但不是类的接口的一部分。例如，类的作者可为方法名附加双下划线前缀，从而指明一个实用方法。本章用私有属性演示了访问方法，并介绍了基本的数据完整性技术。下一章要讨论确保数据完整性的其他技术。下一章讨论的技术允许程序员一方面使用公共访问语法，另一方面又能利用访问方法所提供的数据完整性。这样一来，在程序越变越大、越变越成熟的过程中，程序员可为项目增添数据完整性，同时不必更改项目客户需要依赖的接口。

7.5 为构造函数使用默认参数

迄今为止，都是由客户来提供 `Time` 类的构造函数初始化一个新对象所需的值。但是，构造函数也可定义默认参数，从而在客户没有指定参数的前提下，为对象的属性指定初始值。构造函数还可定义关键字参数，它允许客户只为特定的、命名的参数指定值。图 7.13 (`Time3.py`) 定义了 `Time` 类的一个修改过的版本，它重新定义了 `Time` 构造函数，在其中为每个参数都包含了默认值 0。提供默认构造函数，可保证将对象初始化成一致性状态——即使在构造函数调用中没有提供值。如果程序员提供的构造函数为其所有参数都指定了默认值（或明确不要求参数），就称为“默认构造函数”（换言之，调用它时不必提

供任何参数)。

```

1 # Fig: 7.13: Time3.py
2 # Class Time with default constructor.
3
4 class Time:
5     """Class Time with default constructor"""
6
7     def __init__( self, hour = 0, minute = 0, second = 0 ):
8         """Time constructor initializes each data member to zero"""
9
10        self.setTime( hour, minute, second )
11
12    def setTime( self, hour, minute, second ):
13        """Set values of hour, minute, and second"""
14
15        self.setHour( hour )
16        self.setMinute( minute )
17        self.setSecond( second )
18
19    def setHour( self, hour ):
20        """Set hour value"""
21
22        if 0 <= hour < 24:
23            self.__hour = hour
24        else:
25            raise ValueError, "Invalid hour value: %d" % hour
26
27    def setMinute( self, minute ):
28        """Set minute value"""
29
30        if 0 <= minute < 60:
31            self.__minute = minute
32        else:
33            raise ValueError, "Invalid minute value: %d" % minute
34
35    def setSecond( self, second ):
36        """Set second value"""
37
38        if 0 <= second < 60:
39            self.__second = second
40        else:
41            raise ValueError, "Invalid second value: %d" % second
42
43    def getHour( self ):
44        """Get hour value"""
45
46        return self.__hour
47
48    def getMinute( self ):
49        """Get minute value"""
50
51        return self.__minute
52
53    def getSecond( self ):
54        """Get second value"""
55
56        return self.__second
57
58    def printMilitary( self ):
59        """Prints Time object in military format"""
60
61        print "%.2d:%.2d:%.2d" % \
62            ( self.__hour, self.__minute, self.__second ),
63
64    def printStandard( self ):
65        """Prints Time object in standard format"""
66
67        standardTime = ""
68
```

```

69     if self.__hour == 0 or self.__hour == 12:
70         standardTime += "12:"
71     else:
72         standardTime += "%d:" % ( self.__hour % 12 )
73
74     standardTime += "%.2d:%.2d" % ( self.__minute, self.__second )
75
76     if self.__hour < 12:
77         standardTime += " AM"
78     else:
79         standardTime += " PM"
80
81     print standardTime,

```

图 7.13 为 Time 类定义的默认构造函数

在这个例子中，构造函数使用传给自己的值（或默认值）来调用 setTime 方法。类使用私有属性来存储数据。和 Time 以前的定义一样，setTime 通过类的其他方法来保证提供给 __hour 的值位于 0~23 之间，提供给 __minute 和 __second 的值则位于 0~59 之间。如果值超出范围，方法就会引发一个异常（这是确保数据成员保持一致性状态的一个例子）。

Time 构造函数可包括与 setTime 方法一样的语句。这样做或许更有效，因为避免了对 setTime 的多余调用。然而，如果真的让 Time 构造函数与 setTime 方法完全相同，会增大维护这个类的难度。因为一旦 setTime 方法的实现发生改变，Time 构造函数的实现也要相应改变。但是，对 setTime 的实现任何修改都只需进行一次，由于 Time 构造函数直接调用 setTime，所以减少了修改实现时出现编程错误的可能性。

软件工程知识 7.12 如果类的一个方法提供了构造函数（或其他方法）需要的全部或部分功能，请从构造函数（或其他方法）中调用那个方法。这样可简化代码的维护，并减少代码的实现发生改变后出错的可能。一个通用的规则是：避免重复代码！

图 7.14 初始化 Time 类（定义见图 7.13）的 4 个对象。一个对象直接使用构造函数调用中的 3 个默认参数；一个自己指定了 1 个参数；一个自己指定了 2 个参数；另一个则自己指定了 3 个参数。初始化之后，每个对象属性的值都通过调用 printTimeValue 显示出来（第 6~10 行）。

如果没有为类定义构造函数，解释器会创建一个默认构造函数（它在调用时无需参数）。但是，Python 提供的构造函数不执行任何初始化。所以创建对象时，不能保证对象会处于一致性状态。

```

1  # Fig. 7.14: fig07_14.py
2  # Demonstrating default constructor method for class Time.
3
4  from Time3 import Time
5
6  def printTimeValues( timeToPrint ):
7      timeToPrint.printMilitary()
8      print
9      timeToPrint.printStandard()
10     print
11
12     time1 = Time()           # all default
13     time2 = Time( 2 )       # minute, second default
14     time3 = Time( 21, 34 )   # second default
15     time4 = Time( 12, 25, 42 ) # all specified
16
17     print "Constructed with:"
18
19     print "\nall arguments defaulted:"
20     printTimeValues( time1 )
21
22     print "\nhour specified; minute and second defaulted:"
23     printTimeValues( time2 )
24
25     print "\nhour and minute specified; second defaulted:"
26     printTimeValues( time3 )

```

```

27
28 print "\nhour, minute and second specified:"
29 printTimeValues( time4 )

```

```

Constructed with:

all arguments defaulted:
00:00:00
12:00:00 AM

hour specified; minute and second defaulted:
02:00:00
2:00:00 AM

hour and minute specified; second defaulted:
21:34:00
9:34:00 PM

hour, minute and second specified:
12:25:42
12:25:42 PM

```

图 7.14 用默认构造函数创建对象

7.6 析构函数

构造函数是用于初始化新建对象的一个方法。相反，析构函数会在对象销毁（即不存在更多的对象引用）后执行。类可定义名为 `__del__` 的一个特殊方法，它在最后一个对象引用删除或超出作用域之后执行。^④方法本身并不实际销毁对象，它只是执行“终止清理”，然后由解释器回收对象的内存，使内存能被重用。析构函数通常只指定 `self` 参数，并返回 `None`。

对于以前展示的所有类，我们都没有定义 `__del__` 方法。在 C++ 等程序语言中，析构函数通常用于分配和回收内存。Python 为程序员解决了大多数这样的问题，所以通常不在类定义中包括 `__del__`。偶尔，类会定义 `__del__` 来关闭网络或数据库连接，随后再实际地销毁一个对象。本书后文将在适当的地方讨论这些问题。下一节要为类定义 `__del__` 方法，目的是对创建的所有类对象进行计数。

7.7 类属性

类的每个对象都拥有在构造函数中创建的所有属性的拷贝。特定情况下，类的所有对象只能共享属性的一个拷贝。为此要使用“类属性”。类属性是“类范围”的信息（也就是说，它是类的一个属性，而非类的特定对象的属性）。

现在来看一个电脑游戏的例子，它演示了对类范围数据的需要。假定游戏中有 `Martian`（火星星人）和其他太空怪物。每个 `Martian` 都希望表现得勇敢一些，如发现至少还存在另外 4 名 `Martian`，就会主动攻击其他太空怪物。如果 `Martian` 总数少于 5 名，每个 `Martian` 都会变得非常胆怯。为此，每个 `Martian` 都必须知道 `martianCount`（火星星人计数）。可以为 `Martian` 类的每个对象都设置一个 `martianCount` 属性。但这样一来，每个 `Martian` 都有属性的一个单独拷贝。另外，每次创建一个 `Martian`，都必须更新每个 `Martian` 中的 `martianCount`。多余的拷贝既浪费空间，也浪费时间。相反，我们将 `martianCount` 创建成一个类属性，使其成为类范围的数据。每个 `Martian` 都能将 `martianCount` 视为自己的一个属性，但 Python 实际只需维护 `martianCount` 属性的一个拷贝，从而极大地节省了空间。这种技术还能节省时间；由于只有一个拷贝，所以不必为 `Martian` 类的每个对象都处理单独的 `martianCount` 拷贝。

^④ 实际上，在某些情况下，最后一个对象引用删除之后，并不马上执行 `__del__`。但大多数情况下，都可安全地假定方法会按预期的那样执行。详细信息请参考 www.python.org/doc/current/ref/customization.htm。

性能提示 7.2 如果数据的一个拷贝已经够用，请用类属性以节省空间。

尽管类属性表面上类似于全局变量，但每个类属性都存在于创建它时的那个类的命名空间中。类属性在类定义中只应初始化一次。可通过一个类的任何对象来访问该类的类属性。即使一个类不存在任何对象，该类的类属性也是存在的。在不存在类的对象的时候，为了访问类属性，只需为属性名附加类名前缀，再加一个小数点即可。

软件工程知识 7.13 即使类没有任何对象被实例化，也可以使用类的类属性。

如图 7.15 所示的 `Employee` 类演示了如何定义一个类属性，用它维护该类被实例化的对象计数。在类定义中，类属性 `count` 初始化成 0（第 7 行）。注意，类属性 `count` 是在类定义主体中创建的，而不是在方法内部。语句实际定义了名为 `count` 的一个新变量，值为 0，并将该变量添加到 `Employee` 类的命名空间。

```

1 # Fig. 7.15: EmployeeWithClassAttribute.py
2 # Class Employee with class attribute count.
3
4 class Employee:
5     """Represents an employee"""
6
7     count = 0      # class attribute
8
9     def __init__( self, first, last ):
10         """Initializes firstName, lastName and increments count"""
11
12         self.firstName = first
13         self.lastName = last
14
15         Employee.count += 1    # increment class attribute
16
17         print "Employee constructor for %s, %s" \
18             % ( self.lastName, self.firstName )
19
20     def __del__( self ):
21         """Decrements count and prints message"""
22
23         Employee.count -= 1    # decrement class attribute
24
25         print "Employee destructor for %s, %s" \
26             % ( self.lastName, self.firstName )

```

图 7.15 `Employee` 类的类属性

图 7.16 访问 `Employee` 类的类属性。类属性 `count` 维护着 `Employee` 类的现有对象的计数。而且无论是否存在 `Employee` 类的对象，都能访问它。如果不存在类的对象，程序可通过类名来引用 `count`（第 7 行）。第 10~11 行创建两个 `Employee` 对象。每个 `Employee` 对象创建时，都会调用它的构造函数。注意，在输出中，创建标识符 `employee3`（第 12 行）不会创建 `Employee` 类的一个新对象，所以不会调用 `Employee` 的构造函数。该语句只是为第 18 行创建的对象绑定一个新名称，使 `employee3` 和 `employee1` 引用同一个对象。第 18~20 行使用关键字 `del` 删除对两个 `Employee` 对象的所有引用。除非在第 20 行删除了最后一个对象引用，否则不会执行与第 10 行创建的对象对应的 `__del__` 方法。

```

1 # Fig. 7.16: fig07_16.py
2 # Demonstrating class attribute access.
3
4 from EmployeeWithClassAttribute import Employee
5
6 print "Number of employees before instantiation is", \
7     Employee.count
8
9 # create two Employee objects
10 employee1 = Employee( "Susan", "Baker" )
11 employee2 = Employee( "Robert", "Jones" )

```

```

12 employee3 = employee1
13
14 print "Number of employees after instantiation is", \
15     employee1.count
16
17 # explicitly delete employee objects by removing references
18 del employee1
19 del employee2
20 del employee3
21
22 print "Number of employees after deletion is", \
23     Employee.count

```

```

Number of employees before instantiation is 0
Employee constructor for Baker, Susan
Employee constructor for Jones, Robert
Number of employees after instantiation is 2
Employee destructor for Jones, Robert
Employee destructor for Baker, Susan
Number of employees after deletion is 0

```

图 7.16 类属性——fig07_16.py

7.8 合成：对象引用作为类成员使用

到目前为止，我们定义的类的对象都只有基本类型的属性。有时，程序员需要让对象的属性引用其他类的对象。例如，假定 AlarmClock（闹钟）类的对象需要知道在什么时候闹铃，那么为什么不将 Time 类的一个对象作为 AlarmClock 类的对象的一个成员使用呢？这种功能称为“合成”。

软件工程知识 7.14 合成是软件重用的一种形式，即类的成员引用了其他类的对象。

软件工程知识 7.15 在类的成员引用了另一个类的对象的前提下，使那个成员对象能被公共访问，不但没有违反封装性，而且还可隐藏那个成员对象的私有成员。

图 7.17 定义了 Date 类，图 7.18 则定义了修改过的 Employee 类，它们演示了如何将对象作为其他对象的成员进行引用。其中，Employee 类包含属性 firstName, lastName, birthDate 和 hireDate。birthDate 和 hireDate 是 Date 类的对象，Date 类包含属性 month, day 和 year。图 7.19 的程序实例化 Employee 类的一个对象，然后初始化并显示它的属性。

```

1 # Fig. 7.17: Date.py
2 # Definition of class Date.
3
4 class Date:
5     """Class that represents dates"""
6
7     # class attribute lists number of days in each month
8     daysPerMonth = [
9         0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ]
10
11     def __init__( self, month, day, year ):
12         """Constructor for class Date"""
13
14         if 0 < month <= 12: # validate month
15             self.month = month
16         else:
17             raise ValueError, "Invalid value for month: %d" % month
18
19         if year >= 0: # validate year
20             self.year = year
21         else:
22             raise ValueError, "Invalid value for year: %y" % year
23
24         self.day = self.checkDay( day ) # validate day

```



```

25
26     print "Date constructor:",
27     self.display()
28
29 def __del__( self ):
30     """Prints message when called"""
31
32     print "Date object about to be destroyed:",
33     self.display()
34
35 def display( self ):
36     """Prints Date information"""
37
38     print "%d/%d/%d" % ( self.month, self.day, self.year )
39
40 def checkDay( self, testDay ):
41     """Validates day of the month"""
42
43     # validate day, test for leap year
44     if 0 < testDay <= Date.daysPerMonth[ self.month ]:
45         return testDay
46     elif self.month == 2 and testDay == 29 and \
47         ( self.year % 400 == 0 or
48           self.year % 100 != 0 and self.year % 4 == 0 ):
49         return testDay
50     else:
51         raise ValueError, "Invalid day: %d for month: %d" % \
52             ( testDay, self.month )

```

图 7.17 成员对象——Date.py

```

1 # Fig. 7.18: EmployeeComposition.py
2 # Definition of Employee class with composite members.
3
4 from Date import Date
5
6 class Employee:
7     """Employee class with Date attributes"""
8
9     def __init__( self, firstName, lastName, birthMonth,
10                  birthDay, birthYear, hireMonth, hireDay, hireYear ):
11         """Constructor for class Employee"""
12
13         self.birthDate = Date( birthMonth, birthDay, birthYear )
14         self.hireDate = Date( hireMonth, hireDay, hireYear )
15
16         self.lastName = lastName
17         self.firstName = firstName
18
19         print "Employee constructor: %s, %s" \
20             % ( self.lastName, self.firstName )
21
22     def __del__( self ):
23         """Called before Employee destruction"""
24
25         print "Employee object about to be destroyed: %s, %s" \
26             % ( self.lastName, self.firstName )
27
28     def display( self ):
29         """Prints employee information"""
30
31         print "%s, %s" % ( self.lastName, self.firstName )
32         print "Hired:",
33         self.hireDate.display()
34         print "Birth date:",
35         self.birthDate.display()

```

图 7.18 成员对象——EmployeeComposition.py

```

1 # Fig. 7.19: fig07_19.py
2 # Demonstrating composition: an object with member objects.

```

```

3
4 from EmployeeComposition import Employee
5
6 employee = Employee( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 )
7 print
8
9 employee.display()
10 print

```

```

Date constructor: 7/24/1949
Date constructor: 3/12/1988
Employee constructor: Jones, Bob

Jones, Bob
Hired: 3/12/1988
Birth date: 7/24/1949

Employee object about to be destroyed: Jones, Bob
Date object about to be destroyed: 3/12/1988
Date object about to be destroyed: 7/24/1949

```

图 7.19 成员对象——fig07_19.py

在图 7.18 中, `Employee` 构造函数 (第 9~20 行) 取得 9 个参数, 分别是 `self`, `firstName`, `lastName`, `birthMonth`, `birthDay`, `birthYear`, `hireMonth`, `hireDay` 和 `hireYear`, 并根据最后 6 个参数创建 `Date` 类的对象。参数 `birthMonth`, `birthDay` 和 `birthYear` 传给对象 `birthDate` 的构造函数, 而参数 `hireMonth`, `hireDay` 和 `hireYear` 传给对象 `hireDate` 的构造函数。`Date` 类和 `Employee` 类都定义了 `__del__` 方法, 分别在 `Date` 类的一个对象或者 `Employee` 类的一个对象被销毁时打印一条消息。

7.9 数据抽象和信息隐藏

正如本章开头所述, 类通常在客户面前隐藏了实现细节。这称为“信息隐藏”。作为信息隐藏的一个例子, 下面要讨论一种名为“堆栈”的数据结构。

可将堆栈想象成一叠盘子。添加一个盘子时, 它肯定放在最顶部 (称为将盘子“压入”堆栈)。类似地, 从中取出一个盘子时, 也肯定从最顶部拿走 (称为使盘子“弹出”堆栈)。堆栈是一种“后入先出” (Last-In First-Out, LIFO) 数据结构。最后一个压入 (插入) 堆栈的项目肯定是从堆栈弹出 (删除) 的第一个项目。

可用列表轻易实现堆栈。而且事实上, Python 列表包含了特定的方法, 程序员利用它们可使列表“表现”为堆栈 (第 22 章还将实现我们自己的 `Stack` 类)。堆栈类的客户无需关心堆栈的实现。客户只知道一旦数据项置入堆栈, 便可按后入先出的方式获取。客户关心的只是堆栈提供的功能, 不关心这些功能具体如何实现。这便是“数据抽象”的概念。即使程序员知道一个类的实现细节, 但他们写的代码也不应依赖这些细节。这样一来, 特定的类 (比如用于实现堆栈及其 `push` 和 `pop` 操作的类) 可用另一个版本替换, 而不会影响系统的其余部分。只要类的服务没有改变 (即在新的类定义中, 每个方法仍有相同的名称, 返回相同类型的值, 并定义相同的参数列表), 系统剩余部分便不受干扰。

高级语言的任务是创建一个便于程序员使用的视图。所有人都能接受的标准视图是不存在的, 这正是存在如此多程序语言的原因。Python 中的面向对象编程不过展示了另一个视图。

大多数程序语言都强调“动作”。在这些语言中, 数据存在的理由是为程序必须采取的动作提供支持; 与动作相比, 数据显得“意义不大”, 并被认为是粗糙的、无序的。在这些语言中, 只有少数内建数据类型, 而且程序员很难创建自己的数据类型。Python 的面向对象编程风格则强调了数据的重要性。在 Python 中, 主要的面向对象编程活动便是创建数据类型 (即“类”), 并表达那些数据类型的各个对象如何进行交互。为使语言能顺利凸显数据的重要性, 程序语言社区需对数据的一些记号法进行规范。这里介绍的规范是“抽象数据类型” (Abstract Data Type, ADT)。如同数 10 年前的结构化编程技术, ADT 今天也获得了人们的高度关注。然而, ADT 并不是用来替代结构化编程的。相反, 它们只是提供了一种

附加的规范，以改进程序开发过程。

以内建整数类型为例，大多数人都把它同数学中的整数联系在一起。但事实上，整数类型只是整数的一种抽象表示。和数学中的整数不同，计算机整数的大小是固定的。例如，某些计算机上的整数类型限制在约负 20 亿到正 20 亿之间。如果计算结果超出范围，就会产生错误，计算机会采取一些特殊方式来响应这个问题（具体取决于计算机）。例如，它可能“悄悄”地生成一个不正确的结果。数学整数则无此问题。因此，计算机整数只是对现实世界的整数的一种近似表示。浮点类型和其他内建类型都存在同样的问题。

我们之前一直都在使用整数类型记号法，现在从一个新的角度来考虑它。整数、浮点和字符串等类型都是抽象数据类型的例子。这些类型用于表示现实世界的记号法，并可在某种程度上，在计算机系统中获得令人满意的精度。

ADT 实际包含两种记号法：一种是“数据表示”；另一种是可对数据采取的“操作”或“运算”。例如在 Python 中，整数包含一个整数值（数据），并提供了加、减、乘、除和取模运算。但是，除以零是未定义的。Python 程序员使用类来实现抽象数据类型。

软件工程知识 7.16 程序员可通过类的机制来创建类型。可设计新类型，并像使用内建类型一样方便地使用它们。这一特点使 Python 成为一种“可扩展语言”。尽管能通过新类型方便地扩展语言，但程序员不能改变基本语言。

我们讨论的另一种抽象数据类型是“队列”，可把它想象成正在排队的人。计算机系统内部使用大量队列。队列为其客户提供了易于理解的行为：客户通过一个“入队”（enqueue）操作，在队列中每次放入一个项目；并通过一个“出队”（dequeue）操作，每次取回一个项目。队列按“先入先出”（First-In First-Out, FIFO）顺序返回项目，即最先插入队列的项目也是最先被删除的。从概念上说，队列可以变得无限长，但真正的队列长度有限。

队列隐藏了一个内部数据表示（跟踪当前排队的项目），并向客户提供了一系列操作（入队和出队）。客户不必关心队列的实现；只需按队列“宣称”的那样进行操作。客户对一个项目进行入队操作时，队列应接收那个项目，并把它置于某种内部的 FIFO 数据结构中。类似地，如果客户希望队列前面的第 n 个项目，队列应从其内部表示中删除项目，并按 FIFO 顺序递送项目（也就是说，在队列中存在时间最长的项目应是下一个出队操作所返回的项目）。

队列 ADT 确保了其内部数据结构的完整性。客户不能直接操纵该数据结构——只有队列 ADT 才能访问它的内部数据。客户只能对数据表示执行允许的操作：ADT 会拒绝它的接口没有提供的操作，具体响应可能是生成一条错误消息、终止执行、引发异常（参见第 12 章）或者忽略操作请求。

7.10 软件重用性

Python 程序员既要考虑创建新类，也要考虑重用标准库中的类。在标准库中，包含了大量预定义的类型。开发者将程序员自定义的类同良好定义的、精心测试的、良好文档化的、可移植的以及广泛可用的标准库的类合并到一起，构造出完整软件。利用这种软件重用性，可快速开发出功能强大的、高质量的软件。如今，“快速应用程序开发”（RAD）是许多人关心的一个主题。

标准库允许 Python 开发者重复使用现有的、经广泛测试的类，更快地生成应用程序。除了能缩短开发时间，标准库的类还可改进程序员对应用程序进行调试和维护的能力，因为使用的是经过实际检验的软件组件。程序员要想顺利使用标准库的类，必须熟悉标准库所提供的丰富功能。

本章讨论了如何定义类，以及如何创建类的对象。新建对象时，类的构造函数会初始化新对象的属性。我们讨论了初始化和修改属性的几种方式——默认构造函数、set 方法以及为无效属性值引发异常。还讨论了数据完整性，客户如何直接访问所有对象属性，如何用单下划线前缀（_）向客户指出不应访问属性，以及如何用双下划线前缀（__）对属性名进行重整，以防止无意的属性访问。Python 的直接属性

访问能力有利于快速应用程序开发，并简化了动态内省；但对于大型软件项目来说，仅仅直接访问是不够的。下一章要讨论类的作者如何在一方面保证数据完整性，另一方面仍然利用直接访问语法。将这种数据完整性功能添加到类后，不会干扰客户用于访问对象数据的接口，所以有效促进了安全性、模块化编程技术和快速开发能力——所有这些都是 Python 程序员希望的。

第8章 自定义类

学习目标

- 理解如何编写可自定义一个类的特殊方法
- 会将对象表示成字符串
- 会利用特殊方法来定义属性访问
- 理解如何重定义（重载）运算符使其与新类结合使用
- 理解何时应该、何时不应该重载运算符
- 理解如何重载序列操作
- 学习如何重载映射操作
- 通过实例加深对自定义类的理解

8.1 概述

第7章介绍了 Python 类的基础知识，并讲解了抽象数据类型（ADT）记号法。我们讨论了对象创建和销毁时，如何执行 `__init__` 和 `__del__` 方法。这两个方法是类定义的特殊方法之一。所谓“特殊方法”，是指在 Python 中具有特殊含义的一个方法。客户为对象执行一个特定的操作时，Python 解释器会调用对象的某个特殊方法。举个例子来说，客户创建类的一个对象时，Python 会调用那个类的 `__init__` 特殊方法。

类的作者可实现特殊方法，从而自定义类的行为。自定义的目的是为类的客户提供一种简单的记号法，以便对类的对象进行操纵。例如在第7章，对类进行操纵是通过向对象发送消息（以方法调用的形式）来实现的。这种方法调用记号法对某些类来说显得过于复杂，尤其是数学类。对于这些类，更好的做法是利用 Python 丰富的内建运算符和语句来操纵对象。本章要介绍如何定义特殊方法，允许 Python 运算符与对象配合使用，这称为“运算符重载”。使用这些新功能，可直观而自然地扩展 Python。但运算符重载必须谨慎，因为一旦误用重载功能，会使程序难以理解。

运算符 `+` 在 Python 中具有多种用途，比如整数加法和字符串连接。这就是运算符重载的一个例子。Python 语言本身最起码重用了运算符 `+` 和 `*`。这些运算符会在不同背景下执行最恰当的运算，包括整数运算、浮点运算、字符串处理和其他运算。

Python 允许程序员对大多数运算符进行重载，使其在不同背景下具有不同意义。解释器会根据运算符使用方式采取不同行动。有的运算符经常重载，尤其是运算符 `-` 和 `=`。重载的运算符所执行的工作也可由显式的方法调用来执行，但运算符记号法通常更容易懂。

本章要讨论在什么时候应该使用运算符重载，什么时候则不应该使用。除了展示如何重载运算符之外，我们还要展示使用了重载运算符的完整程序。

自定义还具有其他好处。类可定义特殊方法，使类的一个对象在行为上类似于列表或者字典。类还可定义特殊方法，控制客户通过点访问运算符 `.` 来访问对象属性的方式。本章要介绍合适的特殊方法，并创建用以实现它们的类。

8.2 自定义字符串表示： `__str__` 方法

Python 能用 `print` 语句输出内建数据类型。有时，程序员希望定义一个类，要求它的对象也能用 `print` 语句输出。Python 类可定义特殊方法 `__str__`，为类的对象提供一个不正式的（即人们更容易理解的）字符串表示。如果类的客户程序包含以下语句：

```
print objectOfClass
```

那么 Python 会调用对象的 `__str__` 方法，并输出那个方法所返回的字符串。图 8.1 演示了如何定义特殊方法 `__str__`，用它来处理用户自定义电话号码类 `PhoneNumber` 的数据。程序假定输入的电话号码是正确的。

```

1 # Fig. 8.1: PhoneNumber.py
2 # Representation of phone number in USA format: (xxx) xxx-xxxx.
3
4 class PhoneNumber:
5     """Simple class to represent phone number in USA format"""
6
7     def __init__( self, number ):
8         """Accepts string in form (xxx) xxx-xxxx"""
9
10        self.areaCode = number[ 1:4 ] # 3-digit area code
11        self.exchange = number[ 6:9 ] # 3-digit exchange
12        self.line = number[ 10:14 ] # 4-digit line
13
14    def __str__( self ):
15        """Informal string representation"""
16
17        return "(%s) %s-%s" % \
18            ( self.areaCode, self.exchange, self.line )
19
20    def test():
21
22        # obtain phone number from user
23        newNumber = raw_input(
24            "Enter phone number in the form (123) 456-7890:\n" )
25
26        phone = PhoneNumber( newNumber ) # create PhoneNumber object
27        print "The phone number is:",
28        print phone # invokes phone.__str__()
29
30    if __name__ == "__main__":
31        test()

```

```

Enter phone number in the form (123) 456-7890:
(800) 555-1234
The phone number is: (800) 555-1234

```

图 8.1 字符串表示——特殊方法 `__str__`

方法 `__init__`（第 7~12 行）接收一个形如“(xxx) xxx-xxxx”的字符串。字符串中的每个 x 都是电话号码的一个数位。方法对字符串进行分解，并将电话号码的不同部分作为属性存储。

方法 `__str__`（第 14~18 行）是一个特殊方法，它构造并返回 `PhoneNumber` 类的一个对象的字符串表示。解释器一旦遇到第 28 行的语句：

```
print phone
```

就会执行以下语句：

```
print phone.__str__()
```

程序如果将 `PhoneNumber` 对象传给内建函数 `str`，或者为 `PhoneNumber` 对象使用字符串格式化运算符 `%`（例如 `“%s” % phone`），Python 也会调用 `__str__` 方法。

常见编程错误 8.1 从 `__str__` 方法返回非字符串值是严重的运行时错误。

第 20~28 行的 `test` 函数从用户处请求一个电话号码，新建一个 `PhoneNumber` 对象，并打印对象的字符串表示。记住，假如模块作为一个单独的程序运行（即用户对模块调用 Python 解释器），Python 会将值 `“__main__”` 指派给命名空间的名称（保存在内建变量 `__name__` 中）。如果 `PhoneNumber.py` 作为单独的程序执行，第 31 行就会调用函数 `test`。大多数 Python 模块都采用了这种机制：定义一个 `driver` 函数，检测模块的命名空间以执行函数。这样做的好处在于，模块作者可为模块定义不同的行为，具体由模块

的使用背景决定。如果另一个程序导入模块，`__name__` 的值就会是模块名（例如“PhoneNumber”），而 `test` 函数不会执行。如果模块作为单独的程序执行，`__name__` 的值是“`__main__`”，`test` 函数会执行。第 10 章和第 11 章要创建图形化程序，并用 `test` 函数显示我们定义的图形化组件。

良好编程习惯 8.1 如有必要，请为您创建的模块提供 `test` 函数。这些函数可确保模块正常工作，而且能通过演示模块的工作方式，向客户提供额外的信息。

8.3 自定义属性访问

前一章介绍了客户访问对象属性的两种技术。客户可直接访问属性（使用点访问运算符）；另外，类作者也可将属性指派特殊名称，向客户指明应通过访问方法来访问属性。本节要讨论另一种技术——定义特殊方法，自定义直接属性访问的行为。

Python 提供了一系列特殊方法（参见图 8.2），类可定义这些方法，以控制点访问运算符操纵类对象的方式。重新定义运算符行为的技术称为“运算符重载”，这是后几节要详细讨论的主题。对点访问运算符进行重载，相当于综合了前一章所讨论的两种属性访问技术——客户能直接访问属性（即通过点访问运算符），但这实际是执行访问方法的操作。

方法	说明
<code>__delattr__</code>	客户删除一个属性时执行（例如 <code>del anObject.attribute</code> ）
<code>__getattr__</code>	客户访问一个属性名，但在对象 <code>__dict__</code> 属性中找不到这个名称时执行（例如 <code>anObject.unfoundName</code> ）
<code>__setattr__</code>	客户将值指派给对象的属性时执行（例如 <code>anObject.attribute = value</code> ）

图 8.2 属性访问自定义方法

图 8.3 包含了 `Time` 类的一个修改过的定义，该类在前一章用于探讨属性访问。新的定义通过特殊方法 `__getattr__` 和 `__setattr__` 来控制客户访问及修改对象属性的方式。

第 7~13 行包含 `Time` 类的默认构造函数。构造函数只是将参数值指派给新对象的属性。如果类定义了特殊方法 `__setattr__`，程序每次通过点运算符为对象的属性指派值时，Python 都会调用这个方法。因此，第 11 行的语句实际会生成以下调用：

```

        self.__setattr__( "hour", hour )

1  # Fig: 8.3: TimeAccess.py
2  # Class Time with customized attribute access.
3
4  class Time:
5      """Class Time with customized attribute access"""
6
7      def __init__( self, hour = 0, minute = 0, second = 0 ):
8          """Time constructor initializes each data member to zero"""
9
10         # each statement invokes __setattr__
11         self.hour = hour
12         self.minute = minute
13         self.second = second
14
15     def __setattr__( self, name, value ):
16         """Assigns a value to an attribute"""
17
18         if name == "hour":
19
20             if 0 <= value < 24:
21                 self.__dict__[ "_hour" ] = value
22             else:
23                 raise ValueError, "Invalid hour value: %d" % value
24
```

```

25     elif name == "minute" or name == "second":
26
27         if 0 <= value < 60:
28             self.__dict__[ "_" + name ] = value
29         else:
30             raise ValueError, "Invalid %s value: %d" % \
31                 ( name, value )
32
33     else:
34         self.__dict__[ name ] = value
35
36 def __getattr__( self, name ):
37     """Performs lookup for unrecognized attribute name"""
38
39     if name == "hour":
40         return self._hour
41     elif name == "minute":
42         return self._minute
43     elif name == "second":
44         return self._second
45     else:
46         raise AttributeError, name
47
48 def __str__( self ):
49     """Returns Time object string in military format"""
50     # attribute access does not call __getattr__
51     return "%2d:%2d:%2d" % \
52         ( self._hour, self._minute, self._second )

```

```

Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from TimeAccess import Time
>>> timel = Time( 4, 27, 19 )
>>> print timel
04:27:19
>>> print timel.hour, timel.minute, timel.second
4 27 19
>>> timel.hour = 16
>>> print timel
16:27:19
>>> timel.second = 90
Traceback (most recent call last):
File "<stdin>", line 1, in ?
File "TimeAccess.py", line 30, in __setattr__
raise ValueError, "Invalid %s value: %d" % \
ValueError: Invalid second value: 90

```

图 8.3 自定义属性访问——Time 类

`__setattr__` 方法（第 15~34 行）包含查错代码，用于确保对象数据处于一致性状态。方法要接收 3 个参数：对象引用（`self`）、要设置的属性名以及要指派给属性的值。第 18 行检测要设置的属性是否名叫 "hour"。如果是，第 20~23 行判断指定的值是否在正确的范围中。如果值在正确的范围中，第 21 行访问对象的 `__dict__` 属性中恰当的键-值对，将值指派给属性 `_hour`；否则，第 22~23 行将引发一个异常，指出值是无效的。

非常重要的一点在于，`__setattr__` 方法要用对象的 `__dict__` 属性来设置对象的属性。如果第 21 行包含以下语句：

```
self._hour = value
```

那么 `__setattr__` 方法会再次执行，并使用参数 "hour" 和 `value`，从而导致无穷递归。相反，通过对象的 `__dict__` 属性来指派值，就不是调用 `__setattr__` 方法，而是将相应的键-值对插入对象的 `__dict__`。

常见编程错误 8.2 在 `__setattr__` 方法中，通过点访问运算符为对象属性指派值会造成无穷递归。相反，应使用对象的 `__dict__` 属性。

客户试图为 `minute` 或 `second` 属性指派值时, `__setattr__` 方法的第 25~31 行会执行类似的检测。如果指定的值在正确的范围之内, 方法会将值指派给对象的属性 (`_minute` 或 `_second`)。如果客户试图为 `hour`, `minute` 或 `second` 之外的其他属性指派值, 第 33 行会将值指派给指定的属性名称, 保留 Python 为对象添加属性的默认行为。

常见编程错误 8.3 为对象的属性指派值时, 如果属性名拼写错误, 会造成逻辑错误。Python 将在对象命名空间添加一个新对象, 并采用不正确的名称。

第 36~46 行包含了 `__getattr__` 方法定义。假如客户程序包含以下表达式:

```
time1.attribute
```

并将其作为一个“右值”(运算符右边的值), Python 首先会在 `time1` 的 `__dict__` 属性中查找指定的属性名。如果找到这个名称, Python 将返回属性的值。如果在对象的 `__dict__` 中没有找到属性名, Python 会生成以下调用:

```
time1.__getattr__(attribute)
```

其中的 `attribute` 是客户试图访问的属性名。方法会检测客户是否试图访问 `hour`, `minute` 或者 `second`。如果是, 就返回相应属性的值。否则, 方法会引发一个异常 (第 46 行)。

软件工程知识 8.1 如果找不到属性名, 每个类的 `__getattr__` 定义都应该产生 `AttributeError` 异常, 目的是保留 Python 中用于定位不存在的属性的默认行为。

图 8.3 在类定义之后的交互式会话演示了定义特殊方法 `__getattr__` 和 `__setattr__` 的好处。通过点访问运算符, 客户程序可以以一种透明方式访问 `Time` 类的一个对象的属性。`Time` 类的接口与第 7 章第一个类定义中展示的接口完全一致, 但它现在具有确保数据处于一致性状态的好处。第 9 章要讨论一种类似的技术, 即 `Properties` (属性), 它允许类作者指定一个方法, 以便在客户试图访问或修改一个特定属性时执行这个方法。

软件工程知识 8.2 对于大型系统, 如果需要严格的数据访问, 设计者就应使用 `__getattr__` 和 `__setattr__` 来确保数据的完整性。大型系统的开发者如果使用 Python 2.2, 可借助于 `Properties` 这种更高效技术来利用 `__getattr__` 和 `__setattr__` 所允许的语法。

8.4 运算符重载

利用运算符, 可简洁地表示内建类型的对象处理。还可将运算符用于一个类的对象。尽管 Python 不允许创建新的运算符, 但它允许重载现有的大多数运算符。这样一来, 一旦将这些运算符用于程序员自定义类型的对象, 运算符就会具有与新类型相符的含义。

软件工程知识 8.3 运算符重载促进了 Python 的扩展能力, 这正是 Python 最吸引人的一个特性。

良好编程习惯 8.2 同显式方法调用相比, 如果能使程序更清晰, 就应使用运算符重载完成相同的操作。

良好编程习惯 8.3 避免过度或不一致地使用运算符重载, 因为运算符重载可能使程序条理不清、不易理解。

尽管运算符重载感觉就像一种奇特的功能, 但大多数程序员经常隐式使用重载的运算符。举个例子来说, 加法运算符 (+) 对整数、浮点数和字符串会采取截然不同的操作。但是, 针对这些类型和其他内建类型的变量, 加法仍然能很好地工作, 因为加法运算符 (+) 已在 Python 语言内部进行了重载。

要想重载运算符, 需要和往常一样写一个方法定义, 只是方法名要与用于那个运算符的 Python 特

殊方法对应。例如，方法名`__add__`重载了加法运算符（+）。要将运算符用于类的一个对象，类必须重载那个运算符（即为其定义特殊方法）。

重载尤其适合数学类。它们经常要求重载一系列运算符，目的是与这些数学类在现实世界的处理方式保持一致。例如针对有理数，仅仅重载加法运算符通常是不够的，因为有理数还普遍地使用了其他算术运算符。

Python 提供了丰富的运算符。作为程序员，只有充分理解了每个运算符的含义及背景，才能在为新类重载运算符时做出合理的选择。

运算符重载为用户自定义类提供了简洁的表达式，这类似于 Python 为内建类型提供的丰富的运算符。然而，运算符的重载不是自动进行的。为执行需要的操作，程序员必须编写运算符重载方法。

有可能出现过度重载的情况，比如重载运算符+以执行减法运算，或者重载运算符-以执行乘法运算。像这样不直观地使用重载，会使程序难以理解，所以应尽量避免。

良好编程习惯 8.4 重载运算符并针对对象执行的操作应等同于或类似于运算符针对内建类型的对象执行的操作。

8.5 运算符重载的限制

大多数 Python 运算符和增量赋值符号都能重载^①，如图 8.4 所示。

可以重载的常见运算符和增量赋值语句							
+	-	*	**	/	//	%	<<
>>	&		^	~	<	>	<=
>=	==	!=	+=	-=	*=	**=	/=
//=	%=	<<=	>>=	&=	^=	=	[]
()	.	~	in				

图 8.4 可以重载的运算符和增量赋值语句

不能通过重载来改变运算符的优先级。但是，可在表达式中使用圆括号强制重载运算符的求值顺序。不可通过重载来改变一个运算符的顺序关联性。

不能改变运算符取得的操作数的个数。重载的一元运算符仍然是一元运算符，重载的二元运算符仍然是二元运算符。运算符+和-都有一元和二元版本；这些一元和二元版本可单独重载（使用不同的方法名）。注意，不能创建新的运算符，只能对现有的运算符进行重载。

常见编程错误 8.4 执行重载运算符的方法时，试图通过重载来改变运算符的操作数个数会导致严重的运行时错误。

运算符对内建类型的对象的操作方式不能通过运算符重载加以改变。例如，不能改变运算符+用于对两个整数进行相加的含义。运算符重载只适用于用户自定义类的对象，或者用户自定义类的对象与内建类型的对象的混合形式。

如果重载一元算术运算符（例如+、-和*），会自动重载与运算符对应的增量赋值语句。举个例子来说，如果重载一个加法运算符，从而允许以下形式的语句：

```
object2 = object2 + object1
```

那就表示+=增量赋值语句也会重载，从而允许以下形式的语句：

^① 有两个运算符不能重载，即{}和lambda。lambda 是支持函数式编程的一个关键字，不在本书讨论范围之内。

```
object2 += object1
```

在这个例子中，尽管程序员不必定义一个方法来重载+=赋值语句，但通过为那个类显式地定义方法，也可实现同样的行为。

性能提示 8.1 有时，更好的做法是重载运算符的增量赋值版本，以便能够“当场”执行操作（也就是说，不通过新建对象来占用额外的内存）。

8.6 重载一元运算符

对一个类来说，它的一元运算符要重载为一个方法，它只取得对象引用参数（self）。将一元运算符（比如~）重载为方法后，假定 object1 是 Class 类的一个对象，那么一旦解释器遇到以下表达式：

```
~object1
```

就会生成以下调用：

```
object1.__invert__()
```

操作数 object1 是要为其调用 Class 方法__invert__的对象。图 8.5 总结了一元运算符及其对应的特殊方法。

一元运算符	特殊方法
-	<code>__neg__</code>
+	<code>__pos__</code>
~	<code>__invert__</code>

图 8.5 一元运算符及其对应的特殊方法

8.7 重载二元运算符

对一个类来说，二元运算符或语句要重载为带有两个参数的方法。这两个参数是 self 和 other。本章后面会重载运算符+，以表示 Rational 类的两个对象相加。重载了二元运算符+之后，假定 y 和 z 是 Rational 类的对象，那么 y + z 会被视为 y.__add__(z)，即调用__add__方法。如果 y 不是 Rational 类的对象，但 z 是 Rational 类的对象，那么 y + z 会被视为 z.__radd__(y)。方法被命名为__radd__，是由于要为其执行方法的对象出现在运算符右边。通常，重载的二元运算符方法会创建并返回其对应类的新对象。

如果将赋值语句+=重载为一个 Rational 方法，并接收两个参数，假定 y 和 z 是 Rational 类的对象，那么 y += z 会被视为 y.__iadd__(z)，即调用__iadd__方法。方法被命名为__iadd__，是由于方法要现场（in-place）执行它的操作（也就是说，方法不需占用额外的内存）。通常，这意味着方法要为对象引用参数（self）执行任何必要的计算，再返回更新过的引用。图 8.6 总结了二元运算符和赋值语句，及其对应的特殊方法。

对表达式 y + z 或者语句 y += z 进行求值时，如果只有 y 是 Rational 类的对象，会发生什么情况呢？在这两种情况下，z 必须先转换成 Rational 类的一个对象，然后才能执行相应的运算符重载方法。8.9 节将深入讨论这种转换，以及提供了转换行为的特殊方法。

二元运算符	特殊方法
+	<code>__add__</code> , <code>__radd__</code>
-	<code>__sub__</code> , <code>__rsub__</code>
*	<code>__mul__</code> , <code>__rmul__</code>

二元运算符	特殊方法
/	<code>__div__</code> , <code>__rdiv__</code> , <code>__truediv__</code> (用于 Python 2.2), <code>__rtruediv__</code> (用于 Python 2.2)
//	<code>__floordiv__</code> , <code>__rfloordiv__</code> (用于 Python 2.2)
%	<code>__mod__</code> , <code>__rmod__</code>
**	<code>__pow__</code> , <code>__rpow__</code>
<<	<code>__lshift__</code> , <code>__rlshift__</code>
>>	<code>__rshift__</code> , <code>__rrshift__</code>
&	<code>__and__</code> , <code>__rand__</code>
^	<code>__xor__</code> , <code>__rxor__</code>
	<code>__or__</code> , <code>__ror__</code>
+=	<code>__iadd__</code>
-=	<code>__isub__</code>
*=	<code>__imul__</code>
/=	<code>__idiv__</code> , <code>__itruediv__</code> (用于 Python 2.2)
//=	<code>__ifloordiv__</code> (用于 Python 2.2)
%=	<code>__imod__</code>
*%=	<code>__ipow__</code>
<<=	<code>__ilshift__</code>
>>=	<code>__irshift__</code>
&=	<code>__iand__</code>
^=	<code>__ixor__</code>
.=	<code>__ior__</code>
==	<code>__eq__</code>
!=, <>	<code>__ne__</code>
>	<code>__gt__</code>
<	<code>__lt__</code>
>=	<code>__ge__</code>
<=	<code>__le__</code>

图 8.6 二元运算符及其对应的特殊方法

8.8 重载内建函数

类也可定义某些特殊方法。一旦针对类的一个对象调用特定内建函数，就执行这些方法。例如，可为 `Rational` 类定义特殊方法 `__abs__`，一旦程序调用 `abs(rationalObject)` 来计算那个类的一个对象的绝对值，便执行 `__abs__`。图 8.7 总结了常用的内建函数以及可由类定义的特殊方法。

内建函数	说明	特殊方法
<code>abs(x)</code>	返回 <code>x</code> 的绝对值	<code>__abs__</code>
<code>divmod(x, y)</code>	返回一个元组，其中包含 <code>x % y</code> 的整数商和余数部分	<code>__divmod__</code>
<code>len(x)</code>	返回 <code>x</code> 的长度 (<code>x</code> 应该是一个序列)	<code>__len__</code>
<code>pow(x, y[, z])</code>	返回 <code>x^y</code> 的结果。如果有 3 个参数，就返回 <code>(x^y) % z</code>	<code>__pow__</code>
<code>repr(x)</code>	返回 <code>x</code> 的一个正式字符串表示 (也就是说，可从中复制出对象 <code>x</code> 的一个字符串)	<code>__repr__</code>

图 8.7 常用内建函数及其对应的特殊方法

8.9 类型转换

大多数程序都要处理大量类型的信息。但有时，所有操作都针对一种类型进行。例如，一个字符串与另一个字符串相加（连接）所生成的还是一个字符串。但是，经常都需要将一种类型的数据转换或强制（coerce）为另一种类型。这可能发生在赋值和计算过程中。解释器知道怎样在内建类型之间执行特定的转换。程序员可调用恰当的 Python 函数，比如 `int` 或 `float`，实现内建类型之间的强制转换。

但用户自定义类又该怎么办呢？解释器不知道怎样在用户自定义类和内建类型之间转换。为此，程序员必须采用特殊的方法来覆盖相应的 Python 函数，从而指定此类转换应如何进行。例如，类可定义特殊方法 `__int__`，令其对 `int(anObject)` 调用的行为进行重载，从而返回对象的一个整数表示。图 8.8 的表格总结了为实现类型强制（转换），一个类可以定义的特殊方法。每个特殊方法都有一个对应的内建函数。

方法	说明
<code>__coerce__</code>	将两个值转换成相同类型
<code>__complex__</code>	将对象转换成复数类型
<code>__float__</code>	将对象转换成浮点数类型
<code>__hex__</code>	将对象转换成十六进制字符串类型
<code>__int__</code>	将对象转换成整数类型
<code>__long__</code>	将对象转换成长整数类型
<code>__oct__</code>	将对象转换成八进制字符串类型
<code>__str__</code>	将对象转换成字符串类型。也用于获取对象的非正式字符串表示（即简单描述了对对象的一个字符串）

图 8.8 强制（转换）方法

8.10 案例分析：Rational 类

图 8.9 展示了 `Rational`（有理数）类。它利用重载的数值运算符、内建函数以及语句来处理有理数。所谓有理数，是指表示成分子在上、分母在下的一个分数。有理数可以是正数、负数或者零。`Rational` 类的接口包括一个默认构造函数、字符串表示方法、重载的 `abs` 函数、相等运算符以及几个算术运算符。该类还定义了一个 `simplify` 方法，用于对有理数进行约分。对有理数进行约分的具体过程是：用最大公约数来除分子和分母，从而将有理数表示成“最简形式”。文件定义了一个 `gcd` 函数，`Rational` 类用它计算两个值的最大公约数。

```

1 # Fig. 8.9: RationalNumber.py
2 # Definition of class Rational.
3
4 def gcd( x, y ):
5     """Computes greatest common divisor of two values"""
6
7     while y:
8         z = x
9         x = y
10        y = z % y
11
12    return x
13
14 class Rational:
15     """Representation of rational number"""
16
17     def __init__( self, top = 1, bottom = 1 ):
18         """Initializes Rational instance"""
19 
```

```

20     # do not allow 0 denominator
21     if bottom == 0:
22         raise ZeroDivisionError, "Cannot have 0 denominator"
23
24     # assign attribute values
25     self.numerator = abs( top )
26     self.denominator = abs( bottom )
27     self.sign = ( top * bottom ) / ( self.numerator *
28         self.denominator )
29
30     self.simplify() # Rational represented in reduced form
31
32 # class interface method
33 def simplify( self ):
34     """Simplifies a Rational number"""
35
36     common = gcd( self.numerator, self.denominator )
37     self.numerator /= common
38     self.denominator /= common
39
40 # overloaded unary operator
41 def __neg__( self ):
42     """Overloaded negation operator"""
43
44     return Rational( -self.sign * self.numerator,
45         self.denominator )
46
47 # overloaded binary arithmetic operators
48 def __add__( self, other ):
49     """Overloaded addition operator"""
50
51     return Rational(
52         self.sign * self.numerator * other.denominator +
53         other.sign * other.numerator * self.denominator,
54         self.denominator * other.denominator )
55
56 def __sub__( self, other ):
57     """Overloaded subtraction operator"""
58
59     return self + ( -other )
60
61 def __mul__( self, other ):
62     """Overloaded multiplication operator"""
63
64     return Rational( self.numerator * other.numerator,
65         self.sign * self.denominator *
66         other.sign * other.denominator )
67
68 def __div__( self, other ):
69     """Overloaded / division operator."""
70
71     return Rational( self.numerator * other.denominator,
72         self.sign * self.denominator *
73         other.sign * other.numerator )
74
75 def __truediv__( self, other ):
76     """Overloaded / division operator. (For use with Python
77     versions (>= 2.2) that contain the // operator)"""
78
79     return self.__div__( other )
80
81 # overloaded binary comparison operators
82 def __eq__( self, other ):
83     """Overloaded equality operator"""
84
85     return ( self - other ).numerator == 0
86
87 def __lt__( self, other ):
88     """Overloaded less-than operator"""
89
90     return ( self - other ).sign < 0

```

```

91
92 def __gt__( self, other ):
93     """Overloaded greater-than operator"""
94
95     return ( self - other ).sign > 0
96
97 def __le__( self, other ):
98     """Overloaded less-than or equal-to operator"""
99
100     return ( self < other ) or ( self == other )
101
102 def __ge__( self, other ):
103     """Overloaded greater-than or equal-to operator"""
104
105     return ( self > other ) or ( self == other )
106
107 def __ne__( self, other ):
108     """Overloaded inequality operator"""
109
110     return not ( self == other )
111
112 # overloaded built-in functions
113 def __abs__( self ):
114     """Overloaded built in function abs"""
115
116     return Rational( self.numerator, self.denominator )
117
118 def __str__( self ):
119     """String representation"""
120
121     # determine sign display
122     if self.sign == -1:
123         signString = "-"
124     else:
125         signString = ""
126
127     if self.numerator == 0:
128         return "0"
129     elif self.denominator == 1:
130         return "%s%d" % ( signString, self.numerator )
131     else:
132         return "%s%d/%d" % \
133             ( signString, self.numerator, self.denominator )
134
135 # overloaded coercion capability
136 def __int__( self ):
137     """Overloaded integer representation"""
138
139     return self.sign * divmod( self.numerator,
140         self.denominator )[0]
141
142 def __float__( self ):
143     """Overloaded floating-point representation"""
144
145     return self.sign * float( self.numerator ) / self.denominator
146
147 def __coerce__( self, other ):
148     """Overloaded coercion. Can only coerce int to Rational"""
149
150     if type( other ) == type( 1 ):
151         return ( self, Rational( other ) )
152     else:
153         return None

```

图 8.9 运算符重载——Rational.py

在图 8.9 的类定义中，第 4~12 行定义函数 gcd，它计算两个值的最大公约数。Rational 类用这个函数来约分有理数。

Rational 构造函数（第 17~30 行）取得两个参数，即 top 和 bottom，两者都默认为 1。客户在创建

Rational 类的一个对象时，如试图使分母为 0，构造函数会引发一个异常，即 `ZeroDivisionError`，表明出现了错误（第 21~22 行）。`ZeroDivisionError` 是一个异常对象的名称，解释器启动时，Python 会将该对象放到内建命名空间中。第 12 章将详细讨论它以及其他异常（比如 `IndexError`，`KeyError` 等等）。第 25~26 行将传给构造函数的参数的绝对值指派给对象的分子和分母。第 27~28 行计算并将对象的符号指派给属性 `sign`。第 30 行调用 `simplify` 方法，将有理数约分为最简形式。

`simplify` 方法（第 33~38 行）用于约分 Rational 类的一个对象。该方法首先调用函数 `gcd`，判断得出对象的分子和分母的最大公约数（第 36 行）。然后，方法使用最大公约数对有理数对象进行约分（第 37~38 行）。

`__neg__` 方法（第 41~45 行）对一元否定运算符进行重载。如果 `rational` 是 Rational 类的一个对象，一旦解释器遇到以下表达式：

```
-rational
```

解释器就会生成以下方法调用：

```
rational.__neg__()
```

它会创建 Rational 类的一个新对象，但符号（正负号）与原始对象相反。

`__add__` 方法（第 48~54 行）重载了加法运算符。该方法要取得两个参数：一个是对象引用（`self`）；另一个是对 Rational 类的另一个对象的引用。如果 `rational1` 和 `rational2` 是 Rational 类的两个对象，一旦解释器遇到以下表达式：

```
rational1 + rational2
```

解释器就会生成以下方法调用：

```
rational1.__add__(rational2)
```

该方法调用会创建并返回 Rational 类的一个新对象，它代表 `self` 与 `other` 相加的结果。新值的分子用以下表达式计算：

```
self.sign * self.numerator * other.denominator +
other.sign * other.numerator * self.denominator
```

分母用以下表达式计算：

```
self.denominator * other.denominator
```

`__sub__` 方法（第 56~59 行）重载了二元减法运算符。该方法用重载的运算符+和-创建并返回第一个参数减去第二个参数的结果。

`__mul__` 方法（第 61~66 行）重载了二元乘法运算符。该方法创建并返回 Rational 类的一个新对象，它代表方法的两个参数的乘积。

`__div__` 方法（第 68~73 行）重载了二元除法运算符/，创建并返回 Rational 的一个新对象，代表两个参数相除的结果。`__truediv__` 方法（第 75~79 行）为使用浮点除法的 Python 2.2 和更高版本重载了二元除法运算符/。该方法只是调用 `__div__` 方法，因为/运算符无论如何应执行相同的操作，不管 Python 的版本是多少。注意，在第 2 章中，我们详细讲解了不同 Python 版本的运算符/之间的差异。

`__eq__` 方法（第 82~85 行）重载了二元相等运算符（`==`）。假定 `rational1` 和 `rational2` 是 Rational 类的两个对象，一旦编译器遇到以下表达式：

```
rational == rational2
```

解释器就会生成以下方法调用：


```
rational1.__eq__( rational2 )
```

这个方法会对两个对象执行减法运算，并判断结果的分子是否为 0。Rational 对象在创建时会约分为最简形式，因此，我们不需要先对方法的参数值进行约分，再检测它们是否相等。

`__lt__` 方法（第 87~90 行）重载了二元小于运算符（<）。该方法将其第二个参数从第一个参数中减去，并检测结果的符号是否小于 0。`__gt__` 方法（第 92~95 行）重载了二元大于运算符（>）。该方法将从第一个参数中减去第二个参数，并检测结果是否大于 0。

`__le__`（第 97~100 行）、`__ge__`（第 102~105 行）和 `__ne__`（第 107~110 行）等方法分别为 Rational 的对象重载了 <=、>= 和不相等运算符（!= 和 <>）。这些方法使用重载的相等运算符（==）、大于运算符（>）和小于运算符（<）来执行它们的操作。

第 113~116 行定义了特殊方法 `__abs__`，目的是重载内建的 abs 函数的功能。假定 rational 是 Rational 类的一个对象，一旦解释器遇到以下表达式：

```
abs( rational )
```

解释器就会生成以下方法调用：

```
rational.__abs__()
```

该方法会使用对象引用参数的分子与分母的值（记住，构造函数将这些值存为正整数）来创建 Rational 类的一个新对象。

第 118~133 行定义了 `__str__` 方法，使客户能用 print 语句或内建函数 str 来显示一个 Rational 类的对象的相关信息。如果对象的分子为 0，`__str__` 会返回整数值 0 的字符串表示；如果对象的分母为 1，`__str__` 会返回对象的符号和分子的字符串表示。否则，方法会依次返回对象的符号的字符串表示、对象的分子的字符串表示、一个 "/" 以及对象的分母的字符串表示。

第 136~153 行定义了用于强制（转换）行为的特殊方法。其中，`__int__` 方法（第 136~140 行）会在客户针对 Rational 类的一个对象调用内建函数 int 时执行。方法将调用内建函数 divmod，目的是计算分子除以分母之后，得到的整数商以及余数。从 divmod 返回的元组中的第一个元素会被返回，它代表整数商部分。`__float__` 方法（第 142~145 行）会在客户对 Rational 类的一个对象调用内建函数 float 时执行。方法会将对象的符号（-1 或 1）乘以分子与分母相除的结果，并通过对分子调用 float 函数这一方式，确保返回的是一个浮点值。

`__coerce__` 方法（第 147~153 行）会在客户对 Rational 类的一个对象以及另一个对象调用内建函数 coerce 时执行；或者在客户执行所谓的“混合模式”算术时执行。以下语句是混合模式算术的一个例子：

```
rational + 1
```

它试图将一个整数加到 Rational 类的一个对象上。该语句实际生成了以下方法调用：

```
rational.__add__( rational.__coerce__( 1 ) )
```

特殊方法 `__coerce__` 包含的代码应将对象和其他类型都转换成相同的类型，而且应返回一个元组，其中包含两个转换好的值。用于 Rational 类的 `__coerce__` 方法只转换整数值。第 150 行判断方法的第二个参数的类型是否为整数。如果是，方法就返回一个元组，其中包含对象引用参数以及 Rational 类的一个新对象，后者是通过向 Rational 的构造函数传递整数参数而创建的。Python 会执行特殊方法 `__coerce__`，并在两种类型无法保持一致的前提下返回 None；因此，如果方法的参数不是一个整数，第 153 行会返回 None。

如图 8.10 所示的 driver 程序会创建 Rational 类的对象——rational1 默认初始化为 1/1，rational2 初始化为 10/30，rational3 初始化成 -7/14。Rational 构造函数调用 simplify 方法对指定的分子和分母进行约分。所以，rational2 代表的值是 1/3，而 rational3 代表的值是 -1/2。

```

1 # Fig. 8.10: fig08_10.py
2 # Driver for class Rational.
3
4 from RationalNumber import Rational
5
6 # create objects of class Rational
7 rational1 = Rational() # 1/1
8 rational2 = Rational( 10, 30 ) # 10/30 (reduces to 1/3)
9 rational3 = Rational( -7, 14 ) # -7/14 (reduces to -1/2)
10
11 # print objects of class Rational
12 print "rational1:", rational1
13 print "rational2:", rational2
14 print "rational3:", rational3
15 print
16
17 # test mathematical operators
18 print rational1, "/", rational2, "=", rational1 / rational2
19 print rational3, "-", rational2, "=", rational3 - rational2
20 print rational2, "*", rational3, "=", rational2 * rational3
21 print rational2 * rational3 - rational1
22
23 # overloading + implicitly overloads +=
24 rational1 += rational2 * rational3
25 print "\nrational1 after adding rational2 * rational3:", rational1
26 print
27
28 # test comparison operators
29 print rational1, "<=", rational2, ":", rational1 <= rational2
30 print rational1, ">", rational3, ":", rational1 > rational3
31 print
32
33 # test built-in function abs
34 print "The absolute value of", rational3, "is:", abs( rational3 )
35 print
36
37 # test coercion
38 print rational2, "as an integer is:", int( rational2 )
39 print rational2, "as a float is:", float( rational2 )
40 print rational2, "+ 1 =", rational2 + 1

```

```

rational1: 1
rational2: 1/3
rational3: -1/2

1 / 1/3 = 3
-1/2 - 1/3 = -5/6
1/3 * -1/2 - 1 = -7/6

rational1 after adding rational2 * rational3: 5/6

5/6 <= 1/3 : 0
5/6 > -1/2 : 1

The absolute value of -1/2 is: 1/2

1/3 as an integer is: 0
1/3 as a float is: 0.333333333333
1/3 + 1 = 4/3

```

图 8.10 运算符重载——fig08_10.py

driver 程序使用 print 语句，输出 Rational 类的每个构造好的对象。第 17~21 行演示使用重载的算术运算符 /、- 和 * 的结果。第 24~26 行演示了重载加法运算符 +，会隐式地重载 += 赋值语句。程序用 += 增量赋值语句使 rational1 与 rational2 * rational3 的乘积相加，再打印出结果。然后，程序用重载的比较运算符比较 Rational 类的不同对象，再打印出结果（第 29~31 行）。第 34 行打印 rational3 对象的绝对值。第 38~40 行检测 Rational 的强制（转换）功能，方法是打印整数表示（调用 __int__ 方法）和浮点表示（调

用 `__float__` 方法), 并让 `Rational` 类的一个对象同个整数相加 (调用 `__coerce__` 方法)。

8.11 重载序列运算

前面解释了如何使用特殊方法来定义一个类, 使其具有数值类型的行为。类还可定义几个特殊方法, 以便实现序列运算, 并向其客户提供一个基于列表的接口。类的一个对象可通过下标和分片来开放对其元素的访问; 可将其传给 `len` 函数, 以判断它的长度; 还可以支持列表所支持的运算符和方法。图 8.11 总结了序列类应该提供的一些方法。下节将为一个基于列表的类 (只包含不重复的值) 定义其中的几个方法。

方法	说明
<code>__add__</code> , <code>__radd__</code> , <code>__iadd__</code>	重载加法运算符, 以便连接序列, 例如 <code>sequence1 + sequence2</code>
<code>append</code>	为一个可变序列追加一个元素, 例如 <code>sequence.append(element)</code>
<code>__contains__</code>	检测元素是否为序列成员, 例如 <code>element in sequence</code>
<code>count</code>	判断在一个可变序列中, 元素的出现次数, 例如 <code>sequence.count(element)</code>
<code>__delitem__</code>	从一个可变序列中删除一个项目, 例如 <code>del sequence[index]</code>
<code>__getitem__</code>	根据下标访问元素, 例如 <code>sequence[index]</code>
<code>index</code>	获取元素在可变序列中第一次出现时的索引, 例如 <code>sequence.index(element)</code>
<code>insert</code>	在可变序列的指定索引处插入一个元素, 例如 <code>sequence.insert(index, element)</code>
<code>__len__</code>	判断序列的长度, 例如 <code>len(sequence)</code>
<code>__mul__</code> , <code>__rmul__</code> , <code>__imul__</code>	重载乘法运算符, 以重复序列, 例如 <code>sequence * 3</code>
<code>pop</code>	从可变序列中移除一个元素, 例如 <code>sequence.pop()</code>
<code>remove</code>	从可变序列中移除第一次出现的一个值, 例如 <code>sequence.remove()</code>
<code>reverse</code>	当场反转一个可变序列, 例如 <code>sequence.reverse()</code>
<code>__setitem__</code>	为可变序列赋值, 例如 <code>sequence[index] = value</code>
<code>sort</code>	当场对一个可变序列排序, 例如 <code>sequence.sort()</code>

图 8.11 序列方法

8.12 案例分析: SingleList 类

现在要展示一个示范性的类, 它封装 (包含) 一个列表, 目的是演示如何定义几个特殊方法, 使创建的类具有序列的行为。该列表只允许客户插入新的 (独一无二的) 值, 并可采用表格形式显示列表。这个例子将进一步加深您对数据抽象的理解。另外, 您可考虑如何对这个例子进行改进。类的开发是一个有趣的、富有创造性的以及挑战智力的过程, 务必记住, 您的目标是“创造宝贵的类”。

图 8.12 的程序演示了 `SingleList` 类及其重载的运算符、语句和其他特殊方法。首先要看看 `driver` 程序 (图 8.13), 再来研究类定义以及类的每个方法。

```

1 # Fig. 8.12: NewList.py
2 # Simple class SingleList.
3
4 class SingleList:
5
6     def __init__( self, initialList = None ):
7         """Initializes SingleList instance"""
8
9         self.__list = [] # internal list, contains no duplicates
10
11         # process list passed to __init__, if necessary
12         if initialList:

```

```

13
14     for value in initialList:
15
16         if value not in self.__list:
17             self.__list.append( value ) # add original value
18
19 # string representation method
20 def __str__( self ):
21     """Overloaded string representation"""
22
23     tempString = ""
24     i = 0
25
26     # build output string
27     for i in range( len( self ) ):
28         tempString += "%12d" % self.__list[ i ]
29
30         if ( i + 1 ) % 4 == 0: # 4 numbers per row of output
31             tempString += "\n"
32
33     if i % 4 != 0: # add newline, if necessary
34         tempString += "\n"
35
36     return tempString
37
38 # overloaded sequence methods
39 def __len__( self ):
40     """Overloaded length of the list"""
41
42     return len( self.__list )
43
44 def __getitem__( self, index ):
45     """Overloaded sequence element access"""
46
47     return self.__list[ index ]
48
49 def __setitem__( self, index, value ):
50     """Overloaded sequence element assignment"""
51
52     if value in self.__list:
53         raise ValueError, \
54             "List already contains value %s" % str( value )
55
56     self.__list[ index ] = value
57
58 # overloaded equality operators
59 def __eq__( self, other ):
60     """Overloaded == operator"""
61
62     if len( self ) != len( other ):
63         return 0 # lists of different sizes
64
65     for i in range( 0, len( self ) ):
66
67         if self.__list[ i ] != other.__list[ i ]:
68             return 0 # lists are not equal
69
70     return 1 # lists are equal
71
72 def __ne__( self, other ):
73     """Overloaded != and <> operators"""
74
75     return not ( self == other )

```

图 8.12 重载了运算符的 SingleList 类——SingleList.py

```

1 # Fig. 8.13: fig08_13.py
2 # Driver for simple class SingleList.
3
4 from NewList import SingleList

```

```

5
6 def getIntegers():
7     size = int( raw_input( "List size: " ) )
8
9     returnList = [] # the list to return
10
11     for i in range( size ):
12         returnList.append(
13             int( raw_input( "Integer %d: " % ( i + 1 ) ) ) )
14
15     return returnList
16
17 # input and create integers1 and integers2
18 print "Creating integers1..."
19 integers1 = SingleList( getIntegers() )
20
21 print "Creating integers2..."
22 integers2 = SingleList( getIntegers() )
23
24 # print integers1 size and contents
25 print "\nSize of list integers1 is", len( integers1 )
26 print "List:\n", integers1
27
28 # print integers2 size and contents
29 print "\nSize of list integers2 is", len( integers2 )
30 print "List:\n", integers2
31
32 # use overloaded comparison operator
33 print "Evaluating: integers1 != integers2"
34
35 if integers1 != integers2:
36     print "They are not equal"
37
38 print "\nEvaluating: integers1 == integers2"
39
40 if integers1 == integers2:
41     print "They are equal"
42
43 print "integers1[ 0 ] is", integers1[ 0 ]
44 print "Assigning 0 to integers1[ 0 ]"
45 integers1[ 0 ] = 0
46 print "integers1:\n", integers1

```

```

Creating integers1...

```

```

List size: 8
Integer 1: 1
Integer 2: 2
Integer 3: 3
Integer 4: 4
Integer 5: 5
Integer 6: 6
Integer 7: 7
Integer 8: 8

```

```

Creating integers2...

```

```

List size: 10
Integer 1: 9
Integer 2: 10
Integer 3: 11
Integer 4: 12
Integer 5: 13
Integer 6: 14
Integer 7: 15
Integer 8: 16
Integer 9: 17
Integer 10: 18

```

```

Size of list integers1 is 8

```

```

List:

```

1	2	3	4
5	6	7	8

```

Size of list integers2 is 10
List:
      9      10      11      12
     13      14      15      16
     17      18

Evaluating: integers1 != integers2
They are not equal

Evaluating: integers1 == integers2
integers1[ 0 ] is 1
Assigning 0 to integers1[ 0 ]
integers1:
      0      2      3      4
      5      6      7      8

```

图 8.13 重载了运算符的 SingleList 类——fig08_13.py

图 8.13 的程序首先创建 SingleList 类的两个对象（第 18~22 行）。这个类的构造函数会取得一个列表作为参数。为创建这个列表，我们调用函数 getIntegers（第 6~15 行）。函数提示用户输入整数，并返回由这些整数构成的列表。第 25~26 行使用重载的 Python 函数 len 判断 integers1 的长度，并用 print 语句（它隐式地调用了 __str__ 方法）证明构造函数已正确初始化了列表元素。接着，第 29~30 行输出 integers2 的长度和内容。

第 35~41 行检测重载的相等运算符（==）和不相等运算符（!=），首先对以下条件进行求值：

```
integers1 != integers2
```

如两个对象不相等，程序打印一条消息（第 36 行）。类似地，如果两个对象相等，第 41 行打印一条消息。

第 43 行使用重载的下标运算符来引用 integers1[0]。这个下标式名称作为一个“右值”使用，以便打印 integers1[0] 中的值。第 45 行将 integers1[0] 作为赋值语句的“左值”使用，以便将新值 0 指派给 integers1 的元素 0。

知道程序的工作方式之后，接着要研究一下类的方法定义（图 8.12）。第 6~17 行定义类的构造函数。构造函数将 _list 属性初始化成一个空列表。如用户为 initialList 参数指定了一个值，构造函数会将来自 initialList 的所有不重复的元素插入 _list。

第 20~36 行定义了 __str__ 方法，它用于将 IntegerList 类的对象表示成一个字符串。这个方法通过遍历列表中的元素，并采用表格格式对元素进行格式化（每行 4 个元素），从而生成一个字符串（tempString）。第 36 行返回已经格式化的字符串。

第 39~42 行定义 __len__ 方法，它覆盖 Python len 函数。一旦解释器在 driver 程序中遇到表达式：

```
len( integers1 )
```

解释器就会生成以下调用：

```
integers1.__len__()
```

该方法的作用很简单，只返回属性 _list 的长度。

第 44~56 行为类定义了两个重载的下标运算符。一旦解释器在 driver 程序中遇到以下表达式：

```
integers1[ 0 ]
```

就可能生成以下调用：

```
integers1.__getitem__( 0 )
```

目的是调用合适的方法，并返回元素 0 的值（参见 driver 程序的第 43 行）。也可能生成以下调用：

```
integers1.__setitem__( 0, value )
```

目的是设置一个列表元素的值（参见 driver 程序的第 45 行）。如果运算符[]用在右值表达式中，调用的是 `__getitem__` 方法；如果运算符[]用在左值表达式中，调用的则是 `__setitem__` 方法。

`__getitem__` 方法（第 44~47 行）只是返回恰当元素的值。`__setitem__` 方法（第 49~56 行）首先确定列表是否包含新元素。如列表包含新元素，方法会引发一个异常；否则，方法会设置新值。由于 `SingleList` 的方法处理的只是一个基本列表，所以适用于普通列表数据类型的任何越界错误同样适用于我们的 `SingleList` 类型。

在第 59~70 行，为类定义了重载的相等运算符（`==`）。一旦解释器遇到以下表达式：

```
integers1 == integers2
```

就会生成以下调用，从而调用 `__eq__` 方法：

```
integers1.__eq__( integers2 )
```

如果列表长度不同，`__eq__` 方法会立即返回 0（第 62~63 行）。否则，方法会比较每一对元素（第 65~68 行）。如果所有元素都相同，方法返回 1（第 70 行）。一旦发现有元素不同，方法就会立即返回 0（第 68 行）。第 72~75 行定义了方法 `__ne__`，用于检测两个 `NewLists` 是否不相等。该方法用重载的运算符 `==` 判断两个对象是否不等。

`SingleList` 类只定义了图 8.11 中推荐为序列使用的一些方法。读者可自行尝试增添更多的方法。

8.13 重载映射运算

Python 定义了可为客户提供基于映射的接口的几个特殊方法。类的一个对象如果实现了这些方法，客户就可通过下标来访问它的元素；也可将对象传给函数 `len`，以判断对象的长度（即键-值对的数量）；还可支持字典所支持的方法。图 8.14 的表格总结了一个映射类应该提供的方法。下节将展示一个示范类，其中定义了许多这样的方法，目的是为一个基本对象提供字典接口。

方法	说明
<code>clear</code>	从映射中删除所有项目，例如 <code>mapping.clear()</code>
<code>__contains__</code>	检测成员关系；应返回与 <code>has_key</code> 方法相同的值，例如 <code>key in mapping</code>
<code>copy</code>	返回映射的一个浅拷贝，例如 <code>mapping.copy()</code>
<code>__delitem__</code>	从映射中删除一个项目，例如 <code>del mapping[key]</code>
<code>get</code>	获取映射中的一个键的值，例如 <code>mapping.get(key)</code>
<code>__getitem__</code>	通过键进行下标访问，例如 <code>mapping[key]</code>
<code>has_key</code>	判断映射中是否包含一个键，例如 <code>mapping.has_key(key)</code>
<code>items</code>	获取映射中的键-值对的一个列表，例如 <code>mapping.items()</code>
<code>keys</code>	获取映射中的键的一个列表，例如 <code>mapping.keys()</code>
<code>__len__</code>	判断映射的长度，例如 <code>len(mapping)</code>
<code>__setitem__</code>	根据键进行插入或赋值，例如 <code>mapping[key] = value</code>
<code>values</code>	返回映射中的值的一个列表，例如 <code>mapping.values()</code>
<code>update</code>	从另一个映射插入项目，例如 <code>mapping.update(otherMapping)</code>

图 8.14 映射方法

8.14 案例分析：SimpleDictionary 类

记住，类的一个对象拥有一个命名空间，其中包含了标识符及其值。`__dict__` 属性为每个对象都包含

了这种信息。可根据这个事实，为类的每个对象——提供字典接口。图 8.15 展示了 SimpleDictionary 类，它定义了用于实现类的映射行为的特殊方法。

```

1 # Fig. 8.15: NewDictionary.py
2 # Definition of class SimpleDictionary.
3
4 class SimpleDictionary:
5     """Class to make an instance behave like a dictionary"""
6
7     # mapping special methods
8     def __getitem__( self, key ):
9         """Overloaded key-value access"""
10
11         return self.__dict__[ key ]
12
13     def __setitem__( self, key, value ):
14         """Overloaded key-value assignment/creation"""
15
16         self.__dict__[ key ] = value
17
18     def __delitem__( self, key ):
19         """Overloaded key-value deletion"""
20
21         del self.__dict__[ key ]
22
23     def __str__( self ):
24         """Overloaded string representation"""
25
26         return str( self.__dict__ )
27
28     # common mapping methods
29     def keys( self ):
30         """Returns list of keys in dictionary"""
31
32         return self.__dict__.keys()
33
34     def values( self ):
35         """Returns list of values in dictionary"""
36
37         return self.__dict__.values()
38
39     def items( self ):
40         """Returns list of items in dictionary"""
41
42         return self.__dict__.items()

```

图 8.15 映射接口——SimpleDictionary 类

图 8.15 定义的类中，每个方法都会针对对象的 `__dict__` 属性调用恰当的方法。其中，`__getitem__` 方法（第 8~11 行）接受一个键参数，其中包含要从字典中获取的键。第 11 行用 `[]` 运算符从对象的 `__dict__` 获取指定的键。`__setitem__` 方法（第 13~16 行）接受键和值参数。该方法要在对象的 `__dict__` 中插入或更新键-值对。`__delitem__` 方法（第 18~21 行）会在客户用关键字 `del` 从字典中删除一个键-值对时执行。该方法要从对象的 `__dict__` 中删除键-值对。`__str__` 方法（第 23~26 行）返回 SimpleDictionary 类的一个对象的字符串表示，方法是将对象的 `__dict__` 传给内建函数 `str`。`keys` 方法（第 29~32 行）、`values` 方法（第 34~37 行）和 `items` 方法（第 39~42 行）分别针对对象的 `__dict__` 调用恰当的方法，返回各自需要的值。

图 8.16 的 driver 程序创建 SimpleDictionary 类的一个对象，并用 `print` 语句输出对象的值（第 7~8 行）。第 11~13 行用 `[]` 运算符将新值添加到对象上，这里调用的是 `simple.__setitem__` 方法。第 16 行使用关键字 `del` 从对象删除一个元素，这里调用的是 `object.__delitem__` 方法。第 20~22 行调用方法 `keys`、`values` 和 `items`，打印对象存储的键-值对。

```

1 # Fig. 8.16: fig08_16.py
2 # Driver for class SimpleDictionary.

```



```
3
4 from NewDictionary import SimpleDictionary
5
6 # create and print SimpleDictionary object
7 simple = SimpleDictionary()
8 print "The empty dictionary:", simple
9
10 # add values to simple (invokes simple.__setitem__)
11 simple[ 1 ] = "one"
12 simple[ 2 ] = "two"
13 simple[ 3 ] = "three"
14 print "The dictionary after adding values:", simple
15
16 del simple[ 1 ] # remove a value
17 print "The dictionary after removing a value:", simple
18
19 # use mapping methods
20 print "Dictionary keys:", simple.keys()
21 print "Dictionary values:", simple.values()
22 print "Dictionary items:", simple.items()
```

```
The empty dictionary: {}
The dictionary after adding values: {1: 'one', 2: 'two', 3: 'three'}
The dictionary after removing a value: {2: 'two', 3: 'three'}
Dictionary keys: [2, 3]
Dictionary values: ['two', 'three']
Dictionary items: [(2, 'two'), (3, 'three')]
```

图 8.16 映射接口——fig08_16.py

本章介绍了自定义类的概念。类可定义一些特殊方法来提供基于语法的接口。这些特殊方法在 Python 中执行大范围的任务，其中包括字符串表示、属性访问、运算符重载和下标访问等。我们讨论了用于提供每一种行为的方法，并通过 3 个案例分析来演示如何使用这些方法。下一章要讨论继承，程序员可通过继承定义新类，以利用现有类的属性及行为。这种能力是面向对象编程的一项关键优势，因为它能让程序员把注意力集中于类的新行为上。例如，本章调用对象基本的 `__dict__` 属性的方法，从而实现一个字典接口。这种技术虽然可行，但会产生一些冗余代码。通过继承，则可定义一个类，让它“重用”标准字典类型的行为，无需重新显式地定义每个映射方法。

第9章 面向对象编程：继承

学习目标

- 会通过从现有的类继承而创建新类
- 理解继承如何促进软件重用性
- 理解基类和派生类的记号法
- 理解多态性的概念
- 了解从基类 object 继承的类

9.1 概述

本章要讨论“继承”，它是面向对象编程最重要的功能之一。继承是软件重用的一种形式，它通过吸收现有类的属性和行为，并使用新类需要的功能进行覆盖或修饰，从而创建出新类。软件重用缩短了程序开发的时间。它鼓励程序员重用经过检验和调试的高质量软件，从而大大减少了系统运行后才发现问题可能性。这无疑令人非常振奋。

创建新类时，程序员不必编写全新的属性和方法，只需指明新类继承以前定义好的“基类”的属性和方法。新类称为“派生类”。每个派生类本身也可以是未来一些派生类的基类。在“单一继承”中，类只从一个基类派生；但在“多重继承”中，派生类要从多个基类继承。单一继承最直观，通过我们展示的几个例子，读者可以迅速地掌握它。多重继承则超出了本书的范围——我们没有为它提供专门的例子，同时提醒读者，使用这种强大功能时务必小心，事先一定要进行深入的研究。要想更详细地了解 Python 2.2 的多重继承功能，请访问：

www.python.org/2.2/descrintro.html
python.sourceforge.net/peps/pep-0253.html

派生类也可添加自己的属性和方法，因此派生类的对象可能比基类的对象大一些。派生类比它的基类更具体，代表一个更小的对象集合。使用单一继承，派生类刚开始时与基类完全一样。继承的真正优势在于，可在派生类中针对从基类继承而来的特性，进行添加、替换或改进操作。

继承时，派生类的每个对象也可被视为基类的一个对象。利用这种“派生类对象是基类对象”关系，可进行一些有趣的处理。例如，通过继承，可将多个相关的对象顺序送入一个列表，再将列表的每个元素都作为一个基类对象进行处理。这样一来，多个对象就可采取常规方式进行处理。如本章后面所述，这种名为“多态性”的能力是面向对象编程（OOP）最吸引人的地方之一。

多态性使系统的设计与实现变得更易扩展。这样写出来的程序能泛型地处理一个层次结构中现有的所有类的对象（就像处理基类对象那样）。如果在程序开发过程中需要不存在的类，可将其直接添加到程序的泛型部分，只需很少修改，或根本不需要修改——只要那些类是以泛型方式处理的层次结构的一部分。在程序中，只有要求直接了解添加到层次结构的特定类的那些部分，才需要进行修改。借助于多态性，我们编写的程序能采取一种泛型方式，处理大量现有的和尚未指定的类，这些类是相互联系的。继承和多态性是管理软件复杂性的有效技术。

构建软件系统的传统经验表明，大部分代码处理的都是紧密相关的特例。在这种系统中，很难把握“全局”，因为设计者和程序员必须全神贯注于特例。面向对象的编程通过“抽象”来避免这种“一叶障目”的情况。

这里要区分“属于”（is-a）和“具有”（has-a）这两种关系。“属于”即为继承：在这种关系中，派生类的一个对象也“属于”基类的一个对象。“具有”即为“合成”（参见图 7.18）：在这种关系中，一个

对象“具有”对其他类的一个或多个对象的引用，后者是前者的成员。

派生类可访问它的基类的属性和方法。继承的一个问题在于，派生类也许会继承不需要或者不应该明确需要的方法实现。假如一个基类方法实现对于一个派生类来说是不恰当的，可在派生类中使用恰当的实现在“覆盖”（即重新定义）那个方法。

最令人高兴的或许是，新类的记号方式可从现有“类库”中的类继承。各软件机构既可开发自己的类库，也可使用遍布于全球的其他库。最终，软件可主要通过“标准可重用组件”构建，这类似于今天的计算机硬件组装技术，这样便可满足未来开发越来越强大的软件的需要。

9.2 继承：基类和派生类

一个类的对象经常也“属于”另一个类的对象。例如，矩形肯定“属于”四边形（正方形、平行四边形和梯形也属于四边形）。所以，可认为 Rectangle（矩形）类从 Quadrilateral（四边形）类继承。在这个例子中，Quadrilateral 类是基类，Rectangle 类是派生类。矩形是四边形的一种特定类型，但不能说四边形“属于”矩形（例如，还存在平行四边形这样的四边形）。图 9.1 展示了几个简单的继承例子。

基类	派生类
Student	GraduateStudent
	UndergraduateStudent
Shape	Circle
	Triangle
	Rectangle
Loan	CarLoan
	HomeImprovementLoan
	MortgageLoan
Employee	FacultyMember
	StaffMember
Account	CheckingAccount
	SavingsAccounts

图 9.1 继承的例子

其他面向对象程序语言（例如 Smalltalk 和 Java）则采用了不同的术语：在继承中，基类称为“超类”，而派生类称为“子类”。由于继承所生成的派生类的特性通常比基类更多，所以超类和子类的说法有可能令人混淆，鉴于此，本书将避免采用这些术语。

继承构成了树形层次结构。基类和它的派生类形成了一种分级结构关系。类肯定能够单独存在，但假如将一个类放到继承机制中，这个类要么是基类，负责为其他类提供属性和行为；要么是派生类，继承其他类的属性和行为。

现在来开发一个简单的继承层次结构，如图 9.2 所示。普通的大学社区有几千人员，他们都是社区成员。这些人包括员工、学生和校友。员工要么是教学人员，要么是工作人员。教学人员要么是行政人员（比如院长和系主任），要么是普通教员。这便产生了如图 9.2 所示的继承层次结构。注意，有的行政人员还要授课，所以我们用多重继承来构成 AdministratorTeacher 类。由于学生经常要为大学工作，员工也经常要上课，所以还要用多重继承创建名为 EmployeeStudent 的一个类。

另一个继承层次结构是 Shape 层次结构，如图 9.3 所示。现实世界其实有许多继承的例子。只是人们平时很少采用这种方式对现实世界进行归纳。所以在学习继承时，请务必调整一下自己的思维模式。

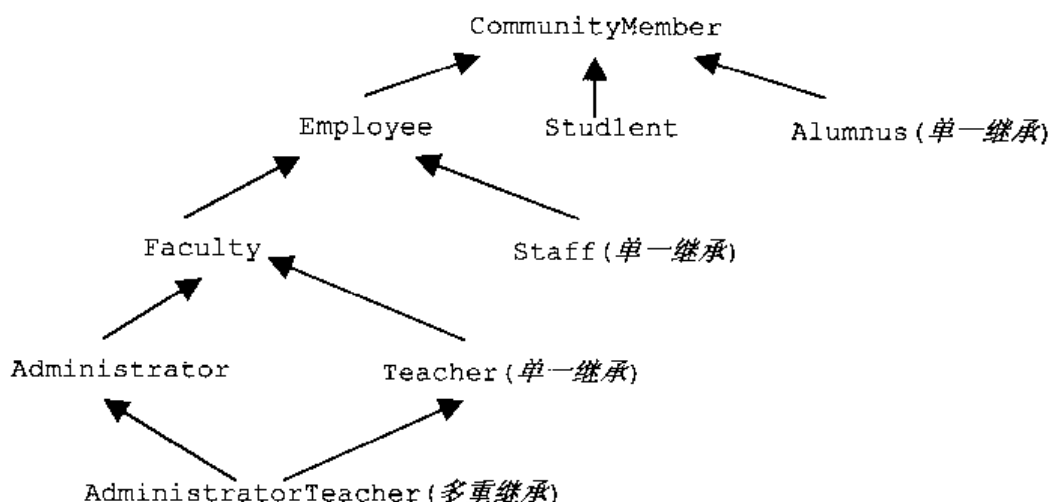


图 9.2 大学社区成员继承结构

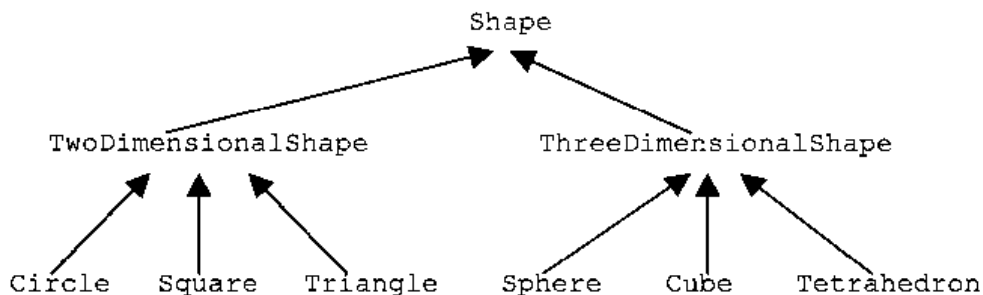


图 9.3 Shape 类层次结构

接着讨论对继承进行描述的语法。要想指明 TwoDimensionalShape（二维形状）类从 Shape 类派生，通常可像下面这样定义 TwoDimensionalShape 类：

```
class TwoDimensionalShape( Shape ):
    ...
```

在这个过程中，基类的所有属性和方法都会作为派生类的属性和方法被继承。

基类可能是一个派生类的“直接基类”，也可能是一个派生类的“间接基类”。定义派生类时，它的直接基类显式地列于圆括号内。定义派生类时，间接基类不会显式地列出。相反，间接基类位于类层次结构向上的两级或更多级上。在图 9.3 中，Circle 类有一个直接基类 TwoDimensionalShape，以及一个间接基类 Shape。尽管 Circle 类的类定义只将 TwoDimensionalShape 列为基类，但 Circle 类会继承 TwoDimensionalShape 和 Shape 类的所有属性和方法。

也许能采用类似的方式对待基类对象和派生类对象；这种共同性是用基类的属性和行为表示的。任何类只要继承自一个公共基类，就可将它的对象当作基类的对象进行处理。9.10 节将讨论一个例子，其中利用了这种关系。

9.3 创建基类和派生类

本节要创建一个继承层次结构，并对来自其中的类的对象进行实例化。Python 提供了两个内建函数，即 `issubclass` 和 `isinstance`，可用它们判断一个类是否派生自另一个类，并判断一个值是特定类的一个对

象，还是那个类的一个子类。图 9.4 运用这些函数演示如何从一个类派生出另一个类，并强调派生类对象“属于”基类对象这一事实。其中，第 6~13 行显示了一个 Point 类及其构造函数定义。第 15~28 行显示了一个 Circle 类和方法定义。第 30~52 行包含一个 driver 程序。这是“结构化继承”的一个例子。尽管表面上不是一系列自然的“属于”关系（许多读者不习惯“圆属于点”这样的说法），但由于我们事实就是从 Point 派生出 Circle，所以“Circle 属于 Point”在理论上是说得通的。本例有助于读者理解继承的机制。本章后面还会展示继承的更自然的例子。

```

1 # Fig 9.4: fig09_04.py
2 # Derived class inheriting from a base class.
3
4 import math
5
6 class Point:
7     """Class that represents geometric point"""
8
9     def __init__( self, xValue = 0, yValue = 0 ):
10         """Point constructor takes x and y coordinates"""
11
12         self.x = xValue
13         self.y = yValue
14
15 class Circle( Point ):
16     """Class that represents a circle"""
17
18     def __init__( self, x = 0, y = 0, radiusValue = 0.0 ):
19         """Circle constructor takes x and y coordinates of center
20         point and radius"""
21
22         Point.__init__( self, x, y ) # call base-class constructor
23         self.radius = float( radiusValue )
24
25     def area( self ):
26         """Computes area of a Circle"""
27
28         return math.pi * self.radius ** 2
29
30 # main program
31
32 # examine classes Point and Circle
33 print "Point bases:", Point.__bases__
34 print "Circle bases:", Circle.__bases__
35
36 # demonstrate class relationships with built-in function issubclass
37 print "\nCircle is a subclass of Point:", \
38     issubclass( Circle, Point )
39 print "Point is a subclass of Circle:", issubclass( Point, Circle )
40
41 point = Point( 30, 50 ) # create Point object
42 circle = Circle( 120, 89, 2.7 ) # create Circle object
43
44 # demonstrate object relationship with built-in function isinstance
45 print "\ncircle is a Point object:", isinstance( circle, Point )
46 print "point is a Circle object:", isinstance( point, Circle )
47
48 # print Point and Circle objects
49 print "\npoint members:\n\t", point.__dict__
50 print "circle members:\n\t", circle.__dict__
51
52 print "\nArea of circle:", circle.area()

```

```

Point bases: ()
Circle bases: (<class __main__.Point at 0x00767250>,)

Circle is a subclass of Point: 1
Point is a subclass of Circle: 0

circle is a Point object: 1

```

```

point is a Circle object: 0
point members:
  {'y': 50, 'x': 30}
circle members:
  {'y': 89, 'x': 120, 'radius': 2.7000000000000002}

Area of circle: 22.9022104447

```

图 9.4 派生类从基类继承

Point 类的构造函数（第 9~13 行）取得两个参数，分别对应点的 x 和 y 坐标。Circle 类（第 15~28 行）从 Point 类继承。类定义第一行的圆括号指明了继承关系。基类的名称（Point）放在圆括号内。Circle 类继承 Point 类的所有属性。这意味着 Circle 类既包含 Point 的成员（即 x 和 y），也包括 Circle 的成员。

派生类要继承其基类中定义的方法，其中包括基类构造函数。通常，派生类会定义一个派生类 `__init__` 方法，从而“覆盖”基类构造函数。如果派生类定义的方法与基类的方法同名，就可以说派生类覆盖了基类方法。覆盖后的派生类构造函数通常会调用基类构造函数，从而先初始化基类属性，再初始化派生类属性。在 Circle 的构造函数中，第 22 行通过一个“非绑定方法调用”来调用基类构造函数。到目前为止，我们进行的都只是“绑定方法调用”。为了进行绑定方法调用，我们要通过一个对象来访问方法名，例如 `anObject.method()`。如前所述，Python 会为绑定方法调用插入对象引用参数。相反，要进行非绑定方法调用，需要通过它的类名来访问方法，并专门传递一个对象引用。例如，第 22 行调用方法 `Point.__init__`，并将 `self`（Circle 类的一个对象）作为对象引用传递。非绑定方法调用还要为 x 和 y 传值，使 Point 构造函数能为 Circle 类的对象初始化 Point 的属性。下一小节将进一步探索方法覆盖以及绑定和非绑定方法调用。基类构造函数终止后，控制权会返回 Circle 构造函数，使其能执行任何 Circle 特有的初始化操作。第 23 行在 Circle 命名空间添加一个新属性，即 `radius`。

软件工程知识 9.1 和其他任何类一样，派生类不一定要定义构造函数。如果派生类没有定义构造函数，一旦客户创建类的一个新对象，就会创建类的基类构造函数。

常见编程错误 9.1 如果派生类的已覆盖构造函数需要调用基类构造函数来初始化基类的成员，派生类构造函数就必须显式地调用基类构造函数。不从派生类中调用基类构造函数是逻辑错误。

常见编程错误 9.2 不将非绑定方法调用的第一个参数指定为对象引用是逻辑错误。

第 25~28 行为 Circle 类定义 `area` 方法。该方法演示了派生类如何定义新方法以扩展基类的功能。在这个例子中，派生类 Circle 提供了额外的功能，以计算 Circle 类的一个对象的面积。

图 9.4 中的 driver 程序首先打印每个类的 `__bases__` 属性值（第 33~34 行）。第 7 章讲过，每个类都具有特殊属性，其中包括 `__bases__`，它是一个元组，包含了对每个类的基类的引用。从输出可以看出，`Point.__bases__` 是一个空元组，因为 Point 不从其他任何类继承。但是 `Circle.__bases__` 元组包含了一个值，即对基类 Point 的一个引用。第 37~39 行调用内建函数 `issubclass`，证明 Circle 是 Point 的一个子类，但 Point 不是 Circle 的一个子类。`issubclass` 函数要取得两个参数（都是类），如果第一个参数代表的类继承自第二个参数代表的类，或者第一个参数是与第二个参数相同的类，就返回 `true`。

第 41~42 行将 `point` 创建成对 Point 类的一个对象的引用，并将 `circle` 创建成对 Circle 类的一个对象的引用。第 45~46 行演示了内建函数 `isinstance`。该函数要取得两个参数：一个对象和一个类。如果对象参数是由类参数所指定的类型的一个对象，或者对象参数是由类参数所指定的类型的一个派生类的对象，`isinstance` 将返回 1。否则，函数返回 0。对 `isinstance` 函数的两次调用证明派生类是其基类的一个对象（例如，`circle` 是一个 Point），但反过来说是不成立的（例如，`point` 不是一个 Circle）。

常见编程错误 9.3 将基类对象当作一个派生类对象处理可能造成运行时错误。如果程序试图从基类对象中调用一个派生类方法，同时基类没有定义那个方法，程序就会终止。

第 49~50 行分别打印 `__dict__` 属性 `point` 和 `circle`。输出结果表明, `circle` 的 `__dict__` 包含属性 `x` 和 `y`, 它们是在基类构造函数中初始化的。第 52 行调用 `circle` 的 `area` 方法, 证明 `Circle` 对功能进行了扩展。

本节演示了基类和派生类的定义过程, 并讨论了绑定和非绑定方法。这些是必须掌握的基本知识, 本章剩余部分将以它们为基础, 对继承和面向对象编程问题展开深入讨论。

9.4 在派生类中覆盖基类方法

派生类可覆盖一个基类方法, 做法是采取相同的名称提供那个方法的一个新版本。如果在派生类中提到了那个方法的名称, 就会选择派生类版本。要从派生类中访问基类版本, 可使用基类的名称, 具体做法是在对基类的方法的一个非绑定调用中, 传递派生类对象。

常见编程错误 9.4 基类方法在派生类中被覆盖后, 经常要让派生类版本调用基类版本, 并执行一些附加的操作。在这种情况下, 如果不使用基类名称来引用基类方法 (也就是为基类方法附加基类名称和一个小数点前缀), 会造成无穷递归, 因为派生类方法实际调用的是自身。这最终导致系统耗尽内存, 造成严重错误。

现在来考虑 `Employee` 类的一个简化版本。它存储员工的 `firstName` 和 `lastName`。这种信息是所有员工 (包括从 `Employee` 类派生的类) 共有的。现在从 `Employee` 类中, 派生出 `HourlyWorker`、`PieceWorker`、`Boss` 和 `CommissionWorker` 类。`HourlyWorker` (钟点工) 按小时计酬, 每周超过 40 小时的每小时按 1.5 个工时计算。`PieceWorker` (计件工) 每生产一件产品, 就计算一件产品的报酬。为简化起见, 假定这个人只生产一种类型的产品, 所以数据成员包括他生产的件数以及每件的报酬。`Boss` (老板) 每周领取固定薪水。`CommissionWorker` (代理人) 每周拿少量固定底薪, 另外抽取每周销售额的一个固定百分比作为自己的佣金。为简化起见, 本节和下一节都只展示 `Employee` 类和派生类 `HourlyWorker`。在 9.10 节, 我们将用一个案例分析来展示完整的继承层次结构。

下一个例子如图 9.5 所示。第 4~16 行展示了 `Employee` 类定义以及 `Employee` 的方法。第 18~40 行展示了 `HourlyWorker` 类定义以及 `HourlyWorker` 的方法定义。第 42~49 行是一个用于 `Employee/HourlyWorker` 继承层次结构的 driver 程序, 它首先创建 `HourlyWorker` 类的一个对象。然后隐式调用其 `__str__` 方法, 随后先通过“绑定方法调用”显式调用该方法, 再通过“非绑定方法调用”显式调用该方法。

```
1 # Fig. 9.5: fig09_05.py
2 # Overriding base-class methods.
3
4 class Employee:
5     """Class to represent an employee"""
6
7     def __init__( self, first, last ):
8         """Employee constructor takes first and last name"""
9
10        self.firstName = first
11        self.lastName = last
12
13    def __str__( self ):
14        """String representation of an Employee"""
15
16        return "%s %s" % ( self.firstName, self.lastName )
17
18    class HourlyWorker( Employee ):
19        """Class to represent an employee paid by hour"""
20
21        def __init__( self, first, last, initHours, initWage ):
22            """Constructor for HourlyWorker, takes first and last name,
23            initial number of hours and initial wage"""
24
25            Employee.__init__( self, first, last )
```



```

26     self.hours = float( initHours )
27     self.wage = float( initWage )
28
29     def getPay( self ):
30         """Calculates HourlyWorker's weekly pay"""
31
32         return self.hours * self.wage
33
34     def __str__( self ):
35         """String representation of HourlyWorker"""
36
37         print "HourlyWorker.__str__ is executing"
38
39         return "%s is an hourly worker with pay of $%.2f" % \
40             ( Employee.__str__( self ), self.getPay() )
41
42 # main program
43 hourly = HourlyWorker( "Bob", "Smith", 40.0, 10.00 )
44
45 # invoke __str__ method several ways
46 print "Calling __str__ several ways..."
47 print hourly # invoke __str__ implicitly
48 print hourly.__str__() # invoke __str__ explicitly
49 print HourlyWorker.__str__( hourly ) # explicit, unbound call

```

```

Calling __str__ several ways...
HourlyWorker.__str__ is executing
Bob Smith is an hourly worker with pay of $400.00
HourlyWorker.__str__ is executing
Bob Smith is an hourly worker with pay of $400.00
HourlyWorker.__str__ is executing
Bob Smith is an hourly worker with pay of $400.00

```

图 9.5 在派生类中覆盖基类方法

Employee 类定义包括两个属性 (firstName 和 lastName) 和两个方法 (__init__ 和 __str__)。构造函数接收两个参数, 并把它们的值指派给 firstName 和 lastName。HourlyWorker 类从 Employee 类继承。HourlyWorker 的成员包括两个属性 (hours 和 wage) 和两个方法 (__init__、getPay 和 __str__)。

HourlyWorker 构造函数通过一个非绑定方法调用, 将 first 和 last 传给 Employee 构造函数, 以便先初始化基类属性, 再初始化 hours 和 wage。getPay 方法使用 hours 和 wage 属性计算 HourlyWorker 的报酬。

HourlyWorker 的 __str__ 方法覆盖了 Employee 的同名方法。人们经常要在派生类中覆盖基类方法, 以扩展其功能。被覆盖的方法有时也要调用该方法的基类版本, 以完成新任务中的一部分。在本例中, 派生类的 __str__ 方法要调用基类的 __str__ 方法 (通过第 40 行的非绑定方法调用) 来输出员工的姓名。派生类的 __str__ 方法还要输出员工的报酬。

driver 程序采取 3 种不同的方式调用 hourly 对象的 __str__ 方法。第 47 行在 print 语句中只使用对象, 这会隐式调用对象 __str__ 方法。第 48 行显式绑定调用对象的 __str__ 方法。第 49 行非绑定调用 HourlyWorker 类的 __str__ 方法, 并将 hourly 作为对象引用参数传递。

9.5 继承的软件工程学

通过继承, 可自定义现有的软件。首先继承现有类的属性和行为, 再添加新的属性或行为, 或覆盖基类行为, 从而对类进行自定义, 使之符合我们的要求。参与大规模软件项目开发的设计人员经常因为面临太多问题而头疼。参与过这种项目的人都知道, 软件重用是改进软件开发过程的关键。包括 Python 在内的面向对象编程则是解决此类问题的良方。

大量有用的模块可充分发挥通过继承实现的软件重用的优势。随着人们对 Python 的兴趣日益增加, 创建有用模块的兴趣也在日益增加。随着个人电脑的问世, 由独立软件供应商所生产的软件制品也呈现

爆炸性增长态势。与此同时，类库的创建和传播也越来越火热。应用程序的设计者可以使用这些库来构建自己的应用程序。另一方面，对库设计者而言，如果他们的产品被封装在他人的应用程序中，也会因此而获得相应的报酬。

软件工程知识 9.2 创建派生类不会影响其基类源代码；基类的完整性通过继承获得了保持。

基类指定的是公共特性——从基类继承的所有类都获得了基类的功能。在面向对象设计过程中，设计者要找出公共特性，并对其进行归纳，以构成合理的基类。然后，派生类自定义除基类功能之外的其他功能。

软件工程知识 9.3 在面向对象系统中，类通常是紧密联系的。因此，请归纳出公共属性和行为，将其放入一个基类，然后通过继承来生成派生类。

在非面向对象系统中，设计者要避免函数的不必要的繁殖。类似地，在面向对象系统中，设计者也要避免类的不必要的繁殖。类繁殖得过多，会造成管理上的问题，并可能妨碍软件重用性，因为类的重用者很难在一个庞大的集合中迅速找到适合自己需要的类。为此，应该创建较少的类，分别提供不同的附加功能。但这样做的缺点在于，对部分用户来说，这种类的功能又显得“过于丰富”。

性能提示 9.1 如果通过继承生成的类超过所需，可能会无谓地浪费内存和处理器资源。因此，请从最能满足您需求的类继承。

注意在阅读一系列派生类定义时，可能容易混淆，因为继承的成员没有显示，但它们仍旧存在于派生类中。派生类的文档也存在类似的问题。

软件工程知识 9.4 派生类包含基类的属性和行为。派生类还可能包含其他属性和行为。

软件工程知识 9.5 修改基类时，只要到基类的接口保持不变，就不必修改派生类。

9.6 合成与继承

前面讨论了“属于”关系，它由继承提供支持。另外还有一种“具有”关系（第7章提供了一个例子），即一个类“具有”对其他类（作为自己的成员使用）的引用。利用这种关系，可通过“合成”现有的类，从而创建出新类。以 Employee、BirthDate 和 TelephoneNumber 这三个类为例，不能说 Employee（员工）“属于”一个 BirthDate（生日），也不能说 Employee“属于”一个 TelephoneNumber（电话号码）。相反，应该说 Employee“具有”一个 BirthDate，以及 Employee“具有”一个 TelephoneNumber。

软件工程知识 9.6 假定一个类是另一个类的成员，只要到成员类的接口保持不变，对成员类进行修改的同时，不需要也对它的封装类进行修改。

9.7 “使用”和“知道”关系

继承和合成使新建的类具有与原有的类大部分共通的内容，从而都有利于软件的重用。还有其他方式可使用类提供的服务。尽管人不是一辆车，人也不包含一辆车，但人肯定可以“使用”一辆车。程序通过引用来调用一个对象的方法，即可“使用”那个对象。

一个对象也可以“知道”另一个对象。知识结构体系普遍运用了这种关系。一个对象可包含对另一个对象的引用，从而“知道”那个对象。在这种情况下，我们说一个对象同另一个对象具有“知道”关系；这有时也称为“关联”（Association）。

9.8 案例分析：Point, Circle 和 Cylinder

本节讨论使用 Point (点)、Circle (圆) 和 Cylinder (圆柱体) 结构化继承层次结构的一个更具体的例子。首先要开发和使用 Point 类 (图 9.6)。接着提供一个例子, 演示如何从 Point 类派生出 Circle 类 (图 9.7)。最后要演示如何再从 Circle 类派生出 Cylinder 类 (图 9.8)。

图 9.6 展示了 Point 类。构造函数 (第 7~11 行) 取得两个参数, 它们对应于点的 x 和 y 坐标。__str__ 方法 (第 13~16 行) 创建 Point 类的一个对象的字符串表示。main 函数中的 driver 程序 (第 19~30 行) 创建 point 对象, 打印它的 x 和 y 属性, 更改属性值, 再打印更改过的 point 对象。

```

1 # Fig 9.6: PointModule.py
2 # Definition and test function for class Point.
3
4 class Point:
5     """Class that represents a geometric point"""
6
7     def __init__( self, xValue = 0, yValue = 0 ):
8         """Point constructor takes x and y coordinates"""
9
10        self.x = xValue
11        self.y = yValue
12
13    def __str__( self ):
14        """String representation of a Point"""
15
16        return "( %d, %d )" % ( self.x, self.y )
17
18 # main program
19 def main():
20     point = Point( 72, 115 ) # create object of class Point
21
22     # print point attributes
23     print "X coordinate is:", point.x
24     print "Y coordinate is:", point.y
25
26     # change point attributes and output new location
27     point.x = 10
28     point.y = 10
29
30     print "The new location of point is:", point
31
32 if __name__ == "__main__":
33     main()

```

```

X coordinate is: 72
Y coordinate is: 115
The new location of point is: ( 10, 10 )

```

图 9.6 Point 类——PointModule.py

图 9.7 展示了 Circle 类, 它从 Point 类继承。第 7~26 行是 Circle 类定义, 第 29~45 行包含 Circle 类的 driver 程序。注意, 由于 Circle 是从 Point 类继承的, 所以 Circle 的接口包含 Point 的方法以及 Circle 自己的 area 方法。

```

1 # Fig. 9.7: CircleModule.py
2 # Definition and test function for class Circle.
3
4 import math
5 from PointModule import Point
6
7 class Circle( Point ):
8     """Class that represents a circle"""
9
10    def __init__( self, x = 0, y = 0, radiusValue = 0.0 ):

```

```

11     """Circle constructor takes center point and radius"""
12
13     Point.__init__( self, x, y ) # call base-class constructor
14     self.radius = float( radiusValue )
15
16     def area( self ):
17         """Computes area of a Circle"""
18
19         return math.pi * self.radius ** 2
20
21     def __str__( self ):
22         """String representation of a Circle"""
23
24         # call base-class __str__ method
25         return "Center = %s Radius = %f" % \
26             ( Point.__str__( self ), self.radius )
27
28 # main program
29 def main():
30     circle = Circle( 37, 43, 2.5 ) # create Circle object
31
32     # print circle attributes
33     print "X coordinate is:", circle.x
34     print "Y coordinate is:", circle.y
35     print "Radius is:", circle.radius
36
37     # change circle attributes and print new values
38     circle.radius = 4.25
39     circle.x = 2
40     circle.y = 2
41
42     print "\nThe new location and radius of circle are:", circle
43     print "The area of circle is: %.2f" % circle.area()
44
45     print "\ncircle printed as a Point is:", Point.__str__( circle )
46
47 if __name__ == "__main__":
48     main()

```

```

X coordinate is: 37
Y coordinate is: 43
Radius is: 2.5

The new location and radius of circle are: Center = ( 2, 2 ) Radius = 4.250000
The area of circle is: 56.75

circle printed as a Point is: ( 2, 2 )

```

图 9.7 Circle 类——CircleModule.py

driver 程序创建 Circle 类的一个对象，然后打印对象的属性。然后，driver 程序更改对象的属性值，再打印更改过的对象。第 43 行调用 circle 对象的 area 方法，显示对象的面积。最后，第 45 行调用 Point 类的 __str__ 方法（作为一个非绑定方法），并将 circle 作为对象引用传递。这个调用的结果是将 Circle 类的对象作为 Point 类的一个对象打印出来，演示如何将派生类对象用作基类对象。

最后一个例子（图 9.8）重用了图 9.6 和图 9.7 的 Point 和 Circle 类定义。第 8~32 行是 Cylinder 类定义，第 35~61 行是 Cylinder 类的 driver 程序。注意 Cylinder 是从 Circle 类继承的，所以 Cylinder 的接口中不仅包括 Circle 和 Point 类的方法，还包括自己特有的 area 方法（覆盖了 Circle 的同名方法）和 volume 方法。Cylinder 构造函数调用了它的直接基类 Circle 的构造函数，但没有调用间接基类 Point 的构造函数。每个派生类构造函数都只能调用该类的直接基类的构造函数。

driver 程序创建 Cylinder 类的一个对象（第 38 行），再打印对象的属性值（第 41~44 行）。然后，driver 程序修改圆柱体的高度、半径和坐标值（第 47~49 行），并输出修改结果（第 50~51 行）。最后，程序向 Point 和 Circle 类的 __str__ 方法发出非绑定方法调用（第 57 和第 61 行），将 Cylinder 类的对象分别作为 Point 和 Circle 类的一个对象打印出来。

这个例子很好地演示了继承。读者现在应已掌握了继承的基础知识。本章后文将介绍如何采取一种常规方式，通过继承来写程序。

```

1 # Fig. 9.8: CylinderModule.py
2 # Definition and test function for class Cylinder.
3
4 import math
5 from PointModule import Point
6 from CircleModule import Circle
7
8 class Cylinder( Circle ):
9     """Class that represents a cylinder"""
10
11     def __init__( self, x, y, radius, height ):
12         """Constructor for Cylinder takes x, y, height and radius"""
13
14         Circle.__init__( self, x, y, radius )
15         self.height = float( height )
16
17     def area( self ):
18         """Calculates (surface) area of a Cylinder"""
19
20         return 2 * Circle.area( self ) + \
21             2 * math.pi * self.radius * self.height
22
23     def volume( self ):
24         """Calculates volume of a Cylinder"""
25
26         return Circle.area( self ) * height
27
28     def __str__( self ):
29         """String representation of a Cylinder"""
30
31         return "%s; Height = %f" % \
32             ( Circle.__str__( self ), self.height )
33
34 # main program
35 def main():
36
37     # create object of class Cylinder
38     cylinder = Cylinder( 12, 23, 2.5, 5.7 )
39
40     # print Cylinder attributes
41     print "X coordinate is:", cylinder.x
42     print "Y coordinate is:", cylinder.y
43     print "Radius is:", cylinder.radius
44     print "Height is:", cylinder.height
45
46     # change Cylinder attributes
47     cylinder.height = 10
48     cylinder.radius = 4.25
49     cylinder.x, cylinder.y = 2, 2
50     print "\nThe new points, radius and height of cylinder are:", \
51         cylinder
52
53     print "\nThe area of cylinder is: %.2f" % cylinder.area()
54
55     # display the Cylinder as a Point
56     print "\ncylinder printed as a Point is:", \
57         Point.__str__( cylinder )
58
59     # display the Cylinder as a Circle
60     print "\ncylinder printed as a Circle is:", \
61         Circle.__str__( cylinder )
62
63 if __name__ == "__main__":
64     main()

```

```

X coordinate is: 12
Y coordinate is: 23
Radius is: 2.5
Height is: 5.7

The new points, radius and height of cylinder are: Center = ( 2, 2 ) Radius = 4.250000; Height
= 10.000000

The area of cylinder is: 380.53

cylinder printed as a Point is: ( 2, 2 )

cylinder printed as a Circle is: Center = ( 2, 2 ) Radius = 4.250000

```

图 9.8 Cylinder 类——CylinderModule.py

9.9 抽象基类和具体类

我们在把类想象成一种类型时，就假定要创建那个类型的对象。但是，偶尔也需要定义一些类（程序员永远不打算创建它的任何对象）。这样的类称为“抽象类”。由于它们在继承层次结构中作为基类使用，所以通常把它们称为“抽象基类”。

我们不为抽象类创建对象。它惟一的用途便是提供一个合适的基类，以便其他类从中继承接口，偶尔也继承它的实现，从中实际创建对象的类称为“具体类”。

可以设计一个名为 `TwoDimensionalShape`（二维形状）的抽象基类，并从中派生出具体类，比如 `Square`（正方形）、`Circle`（圆）和 `Triangle`（三角形）。还可以设计一个名为 `ThreeDimensionalShape`（三维形状）的抽象基类，并从中派生出 `Cube`（立方体）、`Sphere`（球体）和 `Cylinder`（圆柱体）等具体类。抽象基类过于泛型，以至于不能根据它创建真实的对象；要创建对象，我们需要更具体的。这正是具体类的工作，它们提供了更具体的细节，可合理地创建出对象。

层次结构中不一定包含抽象类，但正如后文所述，在许多优秀的面向对象系统中，类层次结构都由一个抽象基类带头。某些情况下，在层次结构顶部的几个级别上，都由抽象类占据。上述层次结构可由抽象基类 `Shape`（形状）带头。在下一级，则可设计两个抽象基类，即 `TwoDimensionalShape` 和 `ThreeDimensionalShape`。在更下一级，就可开始为圆和正方形等二维形状定义具体类，并为球体和圆柱体等三维形状定义具体类。

9.10 案例分析：继承接口和实现

下一个例子将重新考虑 9.4 节引入的 `Employee` 层次结构。这一次，我们要实现完整的类层次结构，让它由抽象基类 `Employee` 带头。`Employee` 的派生类包括 `Boss`，他每周领取固定薪水，不管工作了多少小时；`CommissionWorker`，他每周领取少量固定底薪，另加销售提成；`PieceWorker`，他按生产的件数领取报酬；以及 `HourlyWorker`，他按小时计酬，每周超过 40 小时的每小时按 1.5 个工时计算。

每个具体 `Employee` 类都定义了 `earnings`（收入）方法。该方法调用肯定普遍适用于所有员工。然而，每类员工的收入计算方式有所区别，具体由员工的类决定。这些类全部从基类 `Employee` 派生，所以每个派生类都提供了相应的 `earnings` 实现。要计算任何员工的收入，程序只需针对那个员工的对象调用 `earnings` 方法。

现在来看实际的例子（图 9.9）。首先是 `Employee` 类（第 4~35 行）。这个方法包括一个构造函数，它取得名字（`first`）和姓氏（`last`）作为参数；一个实用方法 `_checkPositive`，它确保属性初始化为正数；以及一个抽象方法 `earnings`。`earnings` 方法调用时会产生一个 `NotImplementedError` 异常（调用的详情将在第 12 章讲述）。这样做的原因很简单，因为在 `Employee` 类中提供该方法的一个实现是没有意义的，我们无法为一名泛型的员工计算收入——事先必须知道员工的类型，才能执行正确的收入计算。在方法

主体中引发一个异常, 就可确保从 Employee 继承的所有类都肯定用更具体的定义覆盖了 earnings 方法。程序员永远都不要试图针对 Employee 基类的一个对象调用该方法。如派生类忘记用恰当的定义来覆盖 earnings 方法, 一旦程序试图从派生类中调用 earnings, 基类中的抽象方法就会引发一个异常。和 earnings 类似, Employee 构造函数也会在程序试图创建抽象基类的一个对象时引发异常。第 11~13 行判断 self 是否为 Employee 类的一个对象, 如果是, 就引发一个相应的异常。

```

1 # Fig 9.9: fig09_09.py
2 # Creating a class hierarchy with an abstract base class.
3
4 class Employee:
5     """Abstract base class Employee"""
6
7     def __init__( self, first, last ):
8         """Employee constructor, takes first name and last name.
9         NOTE: Cannot create object of class Employee."""
10
11         if self.__class__ == Employee:
12             raise NotImplementedError, \
13                 "Cannot create object of class Employee"
14
15         self.firstName = first
16         self.lastName = last
17
18     def __str__( self ):
19         """String representation of Employee"""
20
21         return "%s %s" % ( self.firstName, self.lastName )
22
23     def _checkPositive( self, value ):
24         """Utility method to ensure a value is positive"""
25
26         if value < 0:
27             raise ValueError, \
28                 "Attribute value (%s) must be positive" % value
29         else:
30             return value
31
32     def earnings( self ):
33         """Abstract method; derived classes must override"""
34
35         raise NotImplementedError, "Cannot call abstract method"
36
37 class Boss( Employee ):
38     """Boss class, inherits from Employee"""
39
40     def __init__( self, first, last, salary ):
41         """Boss constructor, takes first and last names and salary"""
42
43         Employee.__init__( self, first, last )
44         self.weeklySalary = self._checkPositive( float( salary ) )
45
46     def earnings( self ):
47         """Compute the Boss's pay"""
48
49         return self.weeklySalary
50
51     def __str__( self ):
52         """String representation of Boss"""
53
54         return "%17s: %s" % ( "Boss", Employee.__str__( self ) )
55
56 class CommissionWorker( Employee ):
57     """CommissionWorker class, inherits from Employee"""
58
59     def __init__( self, first, last, salary, commission, quantity ):
60         """CommissionWorker constructor, takes first and last names,
61         salary, commission and quantity"""
62

```

```

63     Employee.__init__( self, first, last )
64     self.salary = self._checkPositive( float( salary ) )
65     self.commission = self._checkPositive( float( commission ) )
66     self.quantity = self._checkPositive( quantity )
67
68     def earnings( self ):
69         """Compute the CommissionWorker's pay"""
70
71         return self.salary + self.commission * self.quantity
72
73     def __str__( self ):
74         """String representation of CommissionWorker"""
75
76         return "%17s: %s" % ( "Commission Worker",
77                               Employee.__str__( self ) )
78
79 class PieceWorker( Employee ):
80     """PieceWorker class, inherits from Employee"""
81
82     def __init__( self, first, last, wage, quantity ):
83         """PieceWorker constructor, takes first and last names, wage
84         per piece and quantity"""
85
86         Employee.__init__( self, first, last )
87         self.wagePerPiece = self._checkPositive( float( wage ) )
88         self.quantity = self._checkPositive( quantity )
89
90     def earnings( self ):
91         """Compute PieceWorker's pay"""
92
93         return self.quantity * self.wagePerPiece
94
95     def __str__( self ):
96         """String representation of PieceWorker"""
97
98         return "%17s: %s" % ( "Piece Worker",
99                               Employee.__str__( self ) )
100
101 class HourlyWorker( Employee ):
102     """HourlyWorker class, inherits from Employee"""
103
104     def __init__( self, first, last, wage, hours ):
105         """HourlyWorker constructor, takes first and last names,
106         wage per hour and hours worked"""
107
108         Employee.__init__( self, first, last )
109         self.wage = self._checkPositive( float( wage ) )
110         self.hours = self._checkPositive( float( hours ) )
111
112     def earnings( self ):
113         """Compute HourlyWorker's pay"""
114
115         if self.hours <= 40:
116             return self.wage * self.hours
117         else:
118             return 40 * self.wage + ( self.hours - 40 ) * \
119                    self.wage * 1.5
120
121     def __str__( self ):
122         """String representation of HourlyWorker"""
123
124         return "%17s: %s" % ( "Hourly Worker",
125                               Employee.__str__( self ) )
126
127 # main program
128
129 # create list of Employees
130 employees = [ Boss( "John", "Smith", 800.00 ),
131               CommissionWorker( "Sue", "Jones", 200.0, 3.0, 150 ),
132               PieceWorker( "Bob", "Lewis", 2.5, 200 ),
133               HourlyWorker( "Karen", "Price", 13.75, 40 ) ]

```

```

134
135 # print Employee and compute earnings
136 for employee in employees:
137     print "%s earned $%.2f" % (employee, employee.earnings())

```

```

Boss: John Smith earned $800.00
Commission Worker: Sue Jones earned $650.00
Piece Worker: Bob Lewis earned $500.00
Hourly Worker: Karen Price earned $550.00

```

图 9.9 基于抽象类的层次结构

Boss 类 (第 37~54 行) 从 **Employee** 类派生。Boss 的方法包括一个构造函数 (第 40~44 行)、覆盖的 **earnings** 方法 (第 46~49 行) 以及一个 **__str__** 方法 (第 51~54 行)。构造函数 (**__init__** 方法) 取得名字、姓氏和周薪等参数, 并将名字和姓氏传给 **Employee** 构造函数, 以初始化派生类对象的基类部分的 **firstName** 和 **lastName** 成员。**earnings** 方法执行 Boss 特有的收入计算。**__str__** 方法创建一个字符串, 其中含有员工的类型和姓名。

CommissionWorker 类 (第 56~77 行) 从 **Employee** 类派生。它的方法包括一个构造函数 (第 59~66 行)、覆盖的 **earnings** 方法 (第 68~71 行) 以及一个 **__str__** 方法 (第 73~77 行)。构造函数取得名字、姓氏、底薪、单件销售提成和销售产品件数等参数, 并将名字和姓氏传给 **Employee** 构造函数。**earnings** 方法执行 **CommissionWorker** 特有的收入计算。**__str__** 方法创建一个字符串, 其中含有员工类型和姓名。

PieceWorker 类 (第 79~99 行) 从 **Employee** 类派生。它的方法包括一个构造函数 (第 82~88 行)、覆盖的 **earnings** 方法 (第 90~93 行) 以及一个 **__str__** 方法 (第 95~99 行)。构造函数取得名字、姓氏、每件报酬以及生产的产品件数等参数, 并将名字和姓氏传给 **Employee** 构造函数。**earnings** 方法执行 **PieceWorker** 特有的收入计算。**__str__** 方法创建一个字符串, 其中含有员工的类型和姓名。

HourlyWorker 类 (第 101~125 行) 从 **Employee** 类派生。它的方法包括一个构造函数 (第 104~110 行)、覆盖的 **earnings** 方法 (第 112~119 行) 以及一个 **__str__** 方法 (第 121~125 行)。构造函数取得名字、姓氏、每小时报酬和工作小时数等参数, 并将名字和姓氏传给 **Employee** 构造函数。**earnings** 方法执行 **HourlyWorker** 特有的收入计算。**__str__** 方法创建一个字符串, 其中含有员工的类型和姓名。

第 127~137 行是 driver 程序。我们创建 **Employee** 类的 4 个具体对象的一个列表。这些对象包括 **Boss** 类的一个对象, **CommissionWorker** 类的一个对象, **PieceWorker** 类的一个对象, 以及 **HourlyWorker** 类的一个对象。第 136~137 行遍历 **Employee** 类的对象列表, 并针对列表中的每个对象调用 **earnings** 方法。之所以能像这样泛型处理由不同类的对象构成的一个列表, 是因为 Python 具有继承“多态性”行为, 这是下一节要讨论的主题。

9.11 多态性

Python 支持“多态性”——通过继承联系在一起的各个不同类的对象可针对同样的消息 (方法调用) 做出不同的响应。发送给多个类型的对象的相同的消息会呈现出“多种型态”——这正是“多态性”一词的来历。例如, 假定 **Rectangle** (矩形) 类从 **Quadrilateral** (四边形) 类派生, 那么 **Rectangle** “属于” **Quadrilateral** 的一个更具体的版本。能对 **Quadrilateral** 类的一个对象执行的操作 (比如计算周长或面积), 也能对 **Rectangle** 类的一个对象执行。Python 天生便是多态性的, 因为语言采用的是“动态类型定义”机制。这意味着 Python 在运行时判断一个对象是否定义了一个方法或者包含了一个属性。如果是, Python 就调用恰当的方法, 或者访问恰当的属性。另外, 对于不通过继承联系在一起的类的对象, Python 的动态类型定义允许程序对其执行泛型处理。如果一个列表中的对象全部提供相同的操作 (例如, 所有对象都定义了一个特定的方法), 程序可泛型地处理那些对象构成的一个列表。“多态性”术语通常是指通过继承联系在一起的类的对象的行为。因此, 我们要在类继承的背景下讨论多态性。在这个背景下, 层次结构中的所有类都提供了一个公共的接口。

下例使用了图 9.9 的 Employee 基类以及 HourlyWorker 派生类。这两个类分别定义了自己的 `__str__` 方法。如果通过一个 Employee 引用来调用 `__str__` 方法，实际调用的将是 Employee.`__str__`。如果通过一个 HourlyWorker 引用来调用 `__str__` 方法，实际调用的则是 HourlyWorker.`__str__`。派生类也可使用基类的 `__str__` 方法。要从派生类中调用基类的 `__str__` 方法，必须显式地调用它，如下所示：

```
Employee.__str__(hourlyReference)
```

以表明基类的 `__str__` 应该显式调用，并将 `hourlyReference` 作为对象引用参数使用。

通过多态性，方法调用可能导致不同的行动，具体取决于接收调用的那个对象的类。这样便增强了程序员的表示能力。

软件工程知识 9.7 通过多态性，程序员可关注于公共特性，让执行时环境去关心具体特性。程序员可在不知道那些对象是什么类型的情况下，指示大量对象采取恰当的行动。

软件工程知识 9.8 多态性增强了扩展性：软件如果要调用多态行为，编写时便不必关心要向其发送消息的对象的类型。因此可在不修改基本系统的前提下，自由添加新类型对象，令其响应现有的消息。

软件工程知识 9.9 抽象类为类层次结构的各个成员定义了一个接口。抽象类包含的方法将在派生类中定义。在多态性的帮助下，层次结构中的所有方法都可使用同一个接口。

现在讨论一下多态性的应用。一个屏幕管理器需要显示不同类的大量对象，其中包括软件写好之后再加入系统的新类型。系统要能显示各种几何形状（基类是 Shape），例如正方形、圆形、三角形、矩形、点、线等等（都从基类 Shape 派生）。屏幕管理器使用基类引用管理所有需要显示的类。绘制任何对象时（无论它位于继承层次结构的哪一级），屏幕管理器都只是向对象发送一条 draw 消息。draw 方法在每个派生类中都被覆盖。Shape 类的每个对象都知道怎样绘制自己。屏幕管理器不需要关心每个对象的类型，也不需要关心以前是否见过一种类型的对象——它只需告诉每个对象 draw（画）自己就可以了。

多态性尤其适合实现分层式软件系统。例如在操作系统中，每类物理设备的工作方式都是不同的。但无论如何，从设备“读”或“写”数据的命令肯定能统一。发送给设备驱动程序对象的“写”消息需要在那个设备驱动程序的背景下进行专门的解释，这具体取决于设备驱动程序怎样操纵特定类型的设备。但是，就“写”调用本身来说，它和向系统中其他任何设备的“写入”操作没有任何区别——都是将一些数量的字节从内存放到设备中。面向对象的操作系统可使用一个抽象基类提供适用于所有设备驱动程序的一个接口。然后，通过从那个抽象基类继承，派生类可采取类似的方式工作。设备驱动程序具有的功能（即接口）作为抽象基类中的方法提供。在派生类中，对这些方法进行了具体的实现，它们与特定类型的设备驱动程序是对应的。

通过多态性编程，程序可遍历一个容器，比如由类层次结构各个级别上的对象构成的一个列表。例如，由 TwoDimensionalShape 类的对象构成的一个列表可包含来自 Square, Circle, Triangle, Rectangle 和 Line 等派生类的对象。使用多态性，只需发送一条消息要求绘制列表中的每个对象，即可在屏幕上绘制出正确的图像。这个多态性编程的例子突出了包括 Python 在内的自然多态性语言在构建大型的、分层的系统的优势。

9.12 类和 Python 2.2

在 Python 2.2 之前的版本中，类和类型是两种截然不同的编程元素。一方面，类型和类存在区别；另一方面，类又“属于”程序员自定义类型，所以两者出现了冲突。许多 Python 程序员，甚至包括语言的开发者，都不喜欢由于类和类型之间这种无谓差异所带来的限制。例如，由于类型不是类，所以程序员不能从内建类型继承，不便利用列表、字典和其他对象所提供的高级数据处理能力。

自 Python 2.2 起，类的本质与行为都发生了变化，消除了类型和类的差异。在将来的所有 2.x 版本

中，程序员可区分两种不同的类，即所谓的“经典”类（它的行为方式与本章前面以及前两章所展示的类相同）以及“新”类（它们具有新的行为）。Python 2.2 提供了 `object` 类型来定义新类型。直接或间接继承于 `object` 的所有类都具有为新类定义的行为，其中包括许多高级的面向对象特性。本节后文将通过几个例子概括其中的一部分特性。

9.12.1 静态方法

在 Python 2.2 中，所有类（不仅仅是从 `object` 继承的类）都可定义“静态方法”。静态方法可由类的一个客户调用，即使不存在类的任何对象。通常，静态方法是类的一个实用方法，不需要类的一个对象就能执行。图 9.10 演示了一个例子，它重新定义了 `Employee` 类，以提供和员工的工作环境有关的信息。在这个例子中，员工在一个小办公室中工作——只有 10 名员工才能在办公室中舒适地工作。如果同时办公的员工超过 10 名，就显得过于拥挤，员工会觉得不适。`Employee` 类维持着一个类属性，名为 `numberOfEmployees`，它存储着被实例化的 `Employee` 类的对象的数量。类还定义了静态方法 `isCrowded`，它判断员工们是否在一个过分拥挤的环境中工作。

```

1 # Fig. 9.10: EmployeeStatic.py
2 # Class Employee with a static method.
3
4 class Employee:
5     """Employee class with static method isCrowded"""
6
7     numberOfEmployees = 0 # number of Employees created
8     maxEmployees = 10 # maximum number of comfortable employees
9
10    def isCrowded():
11        """Static method returns true if the employees are crowded"""
12
13        return Employee.numberOfEmployees > Employee.maxEmployees
14
15    # create static method
16    isCrowded = staticmethod( isCrowded )
17
18    def __init__( self, firstName, lastName ):
19        """Employee constructor, takes first name and last name"""
20
21        self.first = firstName
22        self.last = lastName
23        Employee.numberOfEmployees += 1
24
25    def __del__( self ):
26        """Employee destructor"""
27
28        Employee.numberOfEmployees -= 1
29
30    def __str__( self ):
31        """String representation of Employee"""
32
33        return "%s %s" % ( self.first, self.last )
34
35 # main program
36 def main():
37     answers = [ "No", "Yes" ] # responses to isCrowded
38
39     employeeList = [] # list of objects of class Employee
40
41     # call static method using class
42     print "Employees are crowded?",
43     print answers[ Employee.isCrowded() ]
44
45     print "\nCreating 11 objects of class Employee..."
46
47     # create 11 objects of class Employee
48     for i in range( 11 ):

```

```

49     employeeList.append( Employee( "John", "Doe" + str( i ) ) )
50
51     # call static method using object
52     print "Employees are crowded?",
53     print answers[ employeeList[ i ].isCrowded() ]
54
55     print "\nRemoving one employee..."
56     del employeeList[ 0 ]
57
58     print "Employees are crowded?", answers[ Employee.isCrowded() ]
59
60 if __name__ == "__main__":
61     main()

```

```

Employees are crowded? No

Creating 11 objects of class Employee...
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? No
Employees are crowded? Yes

Removing one employee...
Employees are crowded? No

```

图 9.10 静态方法——Employee 类

第 4~58 行包含了 Employee 类定义。该类定义了两个类属性：numberOfEmployees（它是已经创建的 Employee 类的对象的数量）和 maxEmployees（它是可在办公室中舒适工作的最大员工数量）。

如果 Employee 类的现有对象数量超过能在办公室中舒适工作的最大员工数，isCrowded 方法（第 10~13 行）就返回 true。方法要通过类名（Employee）访问 numberOfEmployees 和 maxEmployees 这两个类属性。第 16 行指定 isCrowded 是 Employee 类的一个静态方法。一个类如果希望将方法指定为静态的，就必须向内建函数 staticmethod 传递方法的名称，再为函数调用返回的值绑定一个名称。静态方法和普通方法的区别在于，在程序调用静态方法时，Python 不向方法传递对象引用参数。因此，静态方法不将 self 指定为第一个参数。这样一来，即使没有类的对象，也能调用静态方法。

__init__ 方法（第 18~23 行）取得两个参数，对应员工的名字（firstName）和姓氏（lastName）。方法还使 Employee 类的 numberOfEmployees 属性值自增。__del__ 方法（第 25~28 行）使 numberOfEmployees 属性值自减。__str__ 方法（第 30~33 行）返回一个字符串，其中含有员工名字和姓氏。

要调用静态方法，既可使用定义了该方法的那个类的名称，也可使用那个类的一个对象的名称。main 函数（第 36~58 行）演示了客户程序可采用哪些方式来调用一个静态方法。answers 变量（第 37 行）是一个列表，其中包含对于“Are the employees crowded?”（员工拥挤吗）这一问题的可能的答案（“Yes”或“No”）。第 43 行使用类名（Employee）来调用静态方法 isCrowded。方法返回 0，因为没有创建该类的任何对象。第 48~53 行包含一个 for 循环，它创建 Employee 类的 11 个对象，并将每个对象添加到 employeeList 列表中。对于每个对象，程序都用那个类的最新的对象来调用静态方法 isCrowded。针对 isCrowded 的第 11 次调用，程序打印“Yes”，因为现有的员工数量（numberOfEmployees 属性值）超过了能在办公室中舒适工作的最大员工数量（maxEmployees 属性值）。第 56 行从 employeeList 中删除一个对象，它调用了对象的析构函数。第 58 行再次调用静态方法 isCrowded，证明员工数已降至一个可接受的级别。

静态方法在 Java 等语言中至关重要，这些语言要求程序员将所有程序代码都放入一个类定义。使用这些语言，程序员经常要定义只包含静态实用方法的类。随后，类的客户可调用静态实用方法，这和 Python

程序调用模块里定义的函数的方式非常相似。在 Python 中, 静态方法允许程序员更准确地定义一个类接口。如果类的方法不需要类的对象即可执行其的任务, 程序员就可将这个类指定为静态的。

9.12.2 从内建类型继承

新的类行为旨在消除在 Python 2.2 之前的版本中类型和类所存在的隔阂。类和类型统一后, 程序员就可定义一个派生类, 它能从 Python 的某种内建类型 (比如整数、字符串和列表) 中继承, 具体采取的方式和从任何基类继承一个派生类是一样的。在 Python 2.2 中, 解释器把对每种类型的引用放入 `__builtin__` 命名空间。图 9.11 列出了常见的内建类型名称, 可从这些内建类型继承个程序员自定义的类。

类型名称	Python 数据类型
<code>complex</code>	复数
<code>dict</code>	字典
<code>file</code>	文件
<code>float</code>	浮点数
<code>int</code>	整数
<code>list</code>	列表
<code>long</code>	长整数
<code>object</code>	基本对象 (注意, 从 <code>object</code> 继承以创建一个“新”类)
<code>str</code>	字符串
<code>tuple</code>	元组
<code>Unicode</code>	Unicode 字符串

图 9.11 Python 2.2 的内建类型名称

图 9.12 重新定义了 8.12 节的 `SingleList` 类, 该列表只包含独一无二的名称。在第 8 章, `SingleList` 类自己定义了要向客户展示的所有方法。在这个例子中, `SingleList` 则从基类 `list` 继承, 并只覆盖了要在 `SingleList` 类中提供自定义行为的那些方法。该类直接继承了基类 `list` 的其他方法。`list` 的其余方法不必由程序员定义, 即可作为新类接口的一部分提供给客户。

新的 `SingleList` 类 (图 9.12) 将名称 `list` 放在紧跟在类名后的圆括号中, 表明要从基类 `list` 继承。所有内建类型 (`object` 除外) 都从 `object` 继承, 所以从内建类型继承的类 (包括 `SingleList`) 都会显示出“新”类的行为。`SingleList` 类的这个定义有别于以前的定义, 因为该定义并不以一个属性的形式, 维护一个内部的值列表。`SingleList` 就“属于”一个 `list`, 所以类的所有方法都能将对象引用作为一个 `list` 对象处理——不再需要一个额外属性。`SingleList` 类的构造函数 (第 7~14 行) 首先调用基类构造函数以初始化列表。如果客户将一个初始列表值传给类的构造函数, 第 14 行就调用 `SingleList` 的 `merge` 方法 (稍后讨论), 将来自列表参数的独一无二的值添加到由基类构造函数来初始化的空列表中。

```

1 # Fig 9.12: NewList.py
2 # Definition for class SingleList,
3
4 class SingleList( list ):
5
6     # constructor
7     def __init__( self, initialList = None ):
8         """SingleList constructor, takes initial list value.
9         New SingleList object contains only unique values"""
10
11         list.__init__( self )
12
13         if initialList:
14             self.merge( initialList )
15

```

```

16 # utility method
17 def _raiseIfNotUnique( self, value ):
18     """Utility method to raise an exception if value
19     is in list"""
20
21     if value in self:
22         raise ValueError, \
23             "List already contains value %s" % value
24
25 # overloaded sequence operation
26 def __setitem__( self, subscript, value ):
27     """Sets value of particular index. Raises exception if list
28     already contains value"""
29
30     # terminate method on non-unique value
31     self._raiseIfNotUnique( value )
32
33     return list.__setitem__( self, subscript, value )
34
35 # overloaded mathematical operators
36 def __add__( self, other ):
37     """Overloaded addition operator, returns new SingleList"""
38
39     return SingleList( list.__add__( self, other ) )
40
41 def __radd__( self, otherList ):
42     """Overloaded right addition"""
43
44     return SingleList( list.__add__( other, self ) )
45
46 def __iadd__( self, other ):
47     """Overloaded augmented assignment. Raises exception if list
48     already contains any of the values in otherList"""
49
50     for value in other:
51         self.append( value )
52
53     return self
54
55 def __mul__( self, value ):
56     """Overloaded multiplication operator. Cannot use
57     multiplication on SingleLists"""
58
59     raise ValueError, "Cannot repeat values in SingleList"
60
61 # __rmul__ and __imul__ have same behavior as __mul__
62 __rmul__ = __imul__ = __mul__
63
64 # overridden list methods
65 def insert( self, subscript, value ):
66     """Inserts value at specified subscript. Raises exception if
67     list already contains value"""
68
69     # terminate method on non-unique value
70     self._raiseIfNotUnique( value )
71
72     return list.insert( self, subscript, value )
73
74 def append( self, value ):
75     """Appends value to end of list. Raises exception if list
76     already contains value"""
77
78     # terminate method on non-unique value
79     self._raiseIfNotUnique( value )
80
81     return list.append( self, value )
82
83 def extend( self, other ):
84     """Adds to list the values from another list. Raises
85     exception if list already contains value"""
86

```

```

87         for value in other:
88             self.append( value )
89
90     # new SingleList method
91     def merge( self, other ):
92         """Merges list with unique values from other list"""
93
94         # add unique values from other
95         for value in other:
96
97             if value not in self:
98                 list.append( self, value )

```

图 9.12 从内建类型 list 继承——SingleList 类

第 17~23 行定义实用方法 `__raiseIfNotUnique`，它取得准备添加到列表的一个值作为参数，如果列表已经包含那个值，就引发一个异常。在列表中添加新元素的所有 `SingleList` 方法首先调用 `__raiseIfNotUnique` 方法，保证客户只在列表中插入不重复的值。通常，客户程序包含了相应的代码来检测异常，判断值是否成功插入。

`__setitem__` 方法（第 26~33 行）在客户将一个值指派给特定的索引时执行。它首先使用要插入的值调用实用方法 `__raiseIfNotUnique`。如果值已在列表中，实用方法就会引发一个异常，`__setitem__` 会终止，值不会加入列表。相反，如果实用方法没有引发异常，第 33 行就调用基类中的 `__setitem__`，它要么在指定索引位置赋值，要么在索引越界的前提下引发一个异常。

第 36~44 行重载了运算符 `+`，以便在 `SingleList` 出现在运算符左边或右边时执行相加运算。`add_` 和 `__radd__` 方法分别返回 `SingleList` 的一个新对象，它使用传给方法的两个参数的元素进行初始化。这个操作的效果等同于将两个列表合并成一个由不重复的值构成的列表。第 46~53 行重载了增量赋值符号 `+=`。方法现场执行它的操作（即针对对象引用自身执行）。针对右操作数中的每个值，`__iadd__` 方法都会调用 `SingleList` 的 `append` 方法。该方法要么在列表尾部插入一个新值，要么在列表已经包含那个值的前提下，引发一个异常。Python 希望一个重载的增量赋值方法返回类的一个对象（方法就是为这个类定义的），所以第 53 行返回增量对象引用。第 55~62 行为 `SingleList` 类的对象重载乘法运算（即列表重复）。根据定义，`SingleList` 中的任何值都不能重复，所以 `__mul__` 方法会在客户试图采取这样的一个操作时引发异常。第 62 行将 `__rmul__`（右乘法）和 `__imul__`（增量赋值乘法）的名称同为 `__mul__` 定义的方法绑定起来；一旦客户调用这些操作，相应的方法也会引发异常。

第 65~88 行定义方法 `insert`，`append` 和 `extend`，以便将值加入列表。在调用相应的方法的基类版本之前，`insert` 和 `append` 方法首先调用实用方法 `__raiseIfNotUnique`，禁止客户向列表添加重复的值。`extend` 方法使用 `append` 方法将另一个列表的元素添加到引用对象。

第 91~98 行的 `merge` 方法使客户能够将合并 `SingleList` 与可能包含重复值的另一个列表。`merge` 方法具有与基类 `list` 为 `extend` 方法提供的相同的行为。然而，如果客户试图使用一个会在 `SingleList` 中插入重复值的列表来扩展 `SingleList`，派生类中的 `extend` 方法就会引发一个异常。通过提供 `merge` 方法，客户可以在不引发异常的前提下扩展一个 `SingleList`。方法通过为客户提供的列表中的每一个不重复的值调用 `list.append`，确保只将不重复的值添加到 `SingleList`。

图 9.13 的 `driver` 程序既使用了 `SingleList` 特有的功能，也使用了从基类 `list` 继承的功能。第 6~7 行创建并打印 `duplicates` 列表，其中包含重复值。第 9 行创建 `SingleList` 类的一个对象，将 `duplicates` 传给构造函数。`SingleList` 类的新对象 `single` 包含来自 `duplicates` 列表的每一个值。`driver` 程序剩余的部分演示了 `SingleList` 的功能。第 10 行打印 `single`，它隐式调用对象的基类 `__str__` 方法。第 11 行将 `single` 传给函数 `len` 函数，后者调用对象的基类 `__len__` 方法，判断列表中的元素数量。

```

1 # Fig. 9.13: fig09_13.py
2 # Program that uses SingleList
3
4 from NewList import SingleList
5
6 duplicates = [ 1, 2, 2, 3, 4, 3, 6, 9 ]

```

```

7 print "List with duplicates is:", duplicates
8
9 single = SingleList( duplicates ) # create SingleList object
10 print "SingleList, created from duplicates, is:", single
11 print "The length of the list is:", len( single )
12
13 # search for values in list
14 print "\nThe value 2 appears %d times in list" % single.count( 2 )
15 print "The value 5 appears %d times in list" % single.count( 5 )
16 print "The index of 9 in the list is:", single.index( 9 )
17
18 if 4 in single:
19     print "The value 4 was found in list"
20
21 # add values to list
22 single.append( 10 )
23 single += [ 20 ]
24 single.insert( 3, "hello" )
25 single.extend( [ -1, -2, -3 ] )
26 single.merge( [ "hello", 2, 100 ] )
27 print "\nThe list, after adding elements is:", single
28
29 # remove values from list
30 popValue = single.pop()
31 print "\nRemoved", popValue, "from list:", single
32 single.append( popValue )
33 print "Added", popValue, "back to end of list:", single
34
35 # slice list
36 print "\nThe value of single[ 1:4 ] is:", single[ 1:4 ]

```

```

List with duplicates is: [1, 2, 2, 3, 4, 3, 6, 9]
SingleList, created from duplicates, is: [1, 2, 3, 4, 6, 9]
The length of the list is: 6

The value 2 appears 1 times in list
The value 5 appears 0 times in list
The index of 9 in the list is: 5
The value 4 was found in list

The list, after adding elements is: [1, 2, 3, 'hello', 4, 6, 9, 10, 20, -1, -2, -3, 100]

Removed 100 from list: [1, 2, 3, 'hello', 4, 6, 9, 10, 20, -1, -2, -3]
Added 100 back to end of list: [1, 2, 3, 'hello', 4, 6, 9, 10, 20, -1, -2, -3, 100]

The value of single[ 1:4 ] is: [2, 3, 'hello']

```

图 9.13 从内建类型 list 继承——fig09_13.py

第 14~16 行调用 single 的 count 和 index 方法，前者判断特定的元素是否已经包含在列表中，后者在列表中定位一个元素。第 18 行使用关键字 in，它隐式地调用基类的 __contains__ 方法，判断列表中是否包含整数元素 4。第 22~25 行调用覆盖的 SingleList 方法，将元素加入列表。第 22 行调用 append 方法，将一个元素添加到列表。第 23 行用符号 += 追加一个元素，它隐式地调用对象的 __iadd__ 方法。第 24 行调用 insert 方法，将元素 "hello" 插入索引位置 3。第 25 行调用 extend 方法，将来自另一个列表的元素添加到 single 中。所有这些方法都能不重复的元素添加到列表；假如任何一个方法调用试图在列表中添加一个重复值，方法就会引发一个异常（如图 9.14 所示）。第 26 行调用 merge 方法，将 single 中的值与来自另一个列表的值合并到一起。从输出中可以看出，第 26 行的调用的结果是只添加了整数元素 100，因为该元素是 single 尚未包含的惟一的一个值。

图 9.13 的第 30~33 行从列表删除一个元素，将元素添加回列表，再打印结果。这些语句表明，客户可使用基类方法 pop 从列表删除一个值，而且假如重新插入被删除的值，就不会引发一个异常。第 36 行演示 SingleList 从基类 list 中继承了“分片”能力。这充分体现了基于继承的软件重用的巨大优势。在 SingleList 类以前的定义中，我们必须亲自编程以实现这种能力，但在这个版本中，只需从基类继承。


```

Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from NewList import SingleList
>>> single = SingleList( [ 1, 2, 3 ] )
>>>
>>> single.append( 1 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "NewList.py", line 79, in append
    self._raiseIfNotUnique( value )
  File "NewList.py", line 22, in _raiseIfNotUnique
    raise ValueError, \
ValueError: List already contains value 1
>>>
>>> single += [ 2 ]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "NewList.py", line 51, in __iadd__
    self.append( value )
  File "NewList.py", line 79, in append
    self._raiseIfNotUnique( value )
  File "NewList.py", line 22, in _raiseIfNotUnique
    raise ValueError, \
ValueError: List already contains value 2
>>>
>>> single.insert( 0, 1 )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "NewList.py", line 70, in insert
    self._raiseIfNotUnique( value )
  File "NewList.py", line 22, in _raiseIfNotUnique
    raise ValueError, \
ValueError: List already contains value 1
>>>
>>> single.extend( [ 3, 4 ] )
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "NewList.py", line 88, in extend
    self.append( value )
  File "NewList.py", line 79, in append
    self._raiseIfNotUnique( value )
  File "NewList.py", line 22, in _raiseIfNotUnique
    raise ValueError, \
ValueError: List already contains value 3

```

图 9.14 SingleList 类——插入重复值

9.12.3 __getattr__ 方法

第 8 章讨论了 `__getattr__` 方法。一旦客户试图访问一个对象属性，但那个属性名称既不在对象的 `__dict__` 中，又不在对象所属的类的 `__dict__` 中，甚至也不在类的直接或间接基类的 `__dict__` 中，就会执行这个方法。从基类 `object` 继承的类也可定义 `__getattr__` 方法，每次访问属性时都会执行它。图 9.15 展示了一个简单的例子。我们定义了 `DemonstrateAccess` 类（第 4~29 行），它从基类 `object` 继承，并同时提供了 `__getattr__` 和 `__getattribute__` 方法。构造函数创建一个 `value` 属性，并把它初始化成 1。

```

1 # Fig. 9.15: fig09_15.py
2 # Class that defines method __getattribute__
3
4 class DemonstrateAccess( object ):
5     """Class to demonstrate when method __getattribute__ executes"""
6
7     def __init__( self ):

```



```

8      """DemonstrateAccess constructor, initializes attribute
9      value"""
10
11      self.value = 1
12
13      def __getattribute__( self, name ):
14          """Executes for every attribute access"""
15
16          print "__getattribute__ executing..."
17          print "\tClient attempt to access attribute:", name
18
19          return object.__getattribute__( self, name )
20
21      def __getattr__( self, name ):
22          """Executes when client access attribute not in __dict__"""
23
24          print "__getattr__ executing..."
25          print "\tClient attempt to access non-existent attribute:", \
26              name
27
28          raise AttributeError, "Object has no attribute %s" \
29              % name

```

```

Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from fig09_15 import DemonstrateAccess
>>> access = DemonstrateAccess()
>>>
>>> access.value
__getattribute__ executing...
    Client attempt to access attribute: value
1
>>>
>>> access.novalue
__getattribute__ executing...
    Client attempt to access attribute: novalue
__getattr__ executing...
    Client attempt to access non-existent attribute: novalue
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "fig09_15.py", line 28, in __getattr__
    raise AttributeError, "Object has no attribute %s" \
AttributeError: Object has no attribute novalue

```

图 9.15 __getattribute__ 方法以及属性访问

客户每次通过点访问运算符 (.) 来访问一个对象的属性时, `__getattribute__` 方法 (第 13~19 行) 都会执行。方法打印一行文本, 指出方法正在执行, 并用另一行指出客户试图访问的属性名称。第 19 行返回调用基类方法 `__getattribute__` 的结果 (向其传递指定的属性名)。派生类中的 `__getattribute__` 方法必须调用方法的基类版本, 才能获取属性值, 这是由于假如通过对象的 `__dict__` 来访问属性值, 会造成对 `__getattribute__` 的另一次调用。

常见编程错误 9.5 要保证正确的属性访问, `__getattribute__` 方法的一个派生类版本应该调用该方法的基类版本。如试图通过访问对象的 `__dict__` 来返回属性值, 会造成无穷递归。

第 21~29 行定义了 `__getattr__` 方法, 它具有与在“经典”类中相同的行为。换言之, 一旦客户试图访问对象的 `__dict__` 中不包含的属性, 该方法就会执行。方法的输出指出方法正在执行, 并报告客户试图访问的属性名称 (第 24~26 行)。第 28~29 行用于保留 Python 的默认行为——客户访问不存在的属性时, 就引发一个异常。

图 9.15 的交互式会话演示了执行 `__getattribute__` 和 `__getattr__` 方法时的效果。首先创建 `DemonstrateAccess` 类的一个对象, 再使用点运算符访问 `value` 属性。输出结果表明, 为了响应属性访问, 要执行 `__getattribute__` 方法; Python 在交互式会话中显示返回值 (1)。接着, 程序访问不存在的 `novalue`

属性。首先执行的是 `__getattr__` 方法，因为客户每次访问一个属性，该方法都会执行。然后，方法的基类版本判断出对象不包含 `novalue` 属性，所以会改为执行 `__getattr__` 方法。这个方法引发一个异常，指出客户访问的是一个不存在的属性。

9.12.4 `__slots__` 类属性

由于 Python 采取了灵活的设计，所以程序员可编写出能在执行时发生改变的应用程序。这一点对软件开发来说尤其有用。例如在开发期间，图形化应用程序的程序员可创建一个软件，它允许程序员动态改变应用程序的外观（即改变程序的部分代码），同时不要求终止程序。对于 Web 服务器等应用程序来说，这项技术也非常有用，因为这些应用程序需要经常改变，以集成新的特性。但这样的灵活性也有缺陷——从性能上看，灵活的应用程序，或者用灵活语言编写的应用程序，要逊色于那些不灵活的竞争性应用程序。

Python 的灵活性所带来的一个副作用是，在对象创建好之后，程序仍可在对象的命名空间中添加属性。这有时会招致不可预料的结果。举个例子来说，程序员可能在赋值语句中为一个属性名称定义了不正确的类型，但程序不会因此报错。相反，赋值语句会将一个新的属性名及其值绑定到对象，程序会继续执行。为解决这个问题，Python 2.2 允许新类定义一个 `__slots__` 属性，它可列出只允许类的对象拥有的属性。图 9.16 展示了对一个“点”对象的两个简单定义，即 `PointWithoutSlots` 和 `PointWithSlots` 类。程序可为 `PointWithoutSlots` 类的对象添加属性，但不可为 `PointWithSlots` 类的对象添加属性。

```

1  # Fig. 9.16: Slots.py
2  # Simple class with slots
3
4  class PointWithoutSlots:
5      """Programs can add attributes to objects of this class"""
6
7      def __init__( self, xValue = 0.0, yValue = 0.0 ):
8          """Constructor for PointWithoutSlots, initializes x- and
9             y-coordinates"""
10
11         self.x = float( xValue )
12         self.y = float( yValue )
13
14  class PointWithSlots( object ):
15      """Programs cannot add attributes to objects of this class"""
16
17      # PointWithSlots objects can contain only attributes x and y
18      __slots__ = [ "x", "y" ]
19
20      def __init__( self, xValue = 0.0, yValue = 0.0 ):
21          """Constructor for PointWithoutSlots, initializes x- and
22             y-coordinates"""
23
24         self.x = float( xValue )
25         self.y = float( yValue )
26
27  # main program
28  def main():
29      noSlots = PointWithoutSlots()
30      slots = PointWithSlots()
31
32      for point in [ noSlots, slots ]:
33          print "\nProcessing an object of class", point.__class__
34
35          print "The current value of point.x is:", point.x
36          newValue = float( raw_input( "Enter new x coordinate: " ) )
37          print "Attempting to set new x-coordinate value..."
38
39          # Logic error: create new attribute called X, instead of
40          # changing the value of attribute x
41          point.X = newValue

```

```

42
43     # output unchanged attribute x
44     print "The new value of point.x is:", point.x
45
46 if __name__ == "__main__":
47     main()

```

```

Processing an object of class __main__.PointWithoutSlots
The current value of point.x is: 0.0
Enter new x coordinate: 1.0
Attempting to set new x-coordinate value...
The new value of point.x is: 0.0

Processing an object of class <class '__main__.PointWithSlots'>
The current value of point.x is: 0.0
Enter new x coordinate: 1.0
Attempting to set new x-coordinate value...
Traceback (most recent call last):
  File "Slots.py", line 47, in ?
    main()
  File "Slots.py", line 41, in main
    point.X = newValue
AttributeError: 'PointWithSlots' object has no attribute 'X'

```

图 9.16 __slots__ 属性——规定对象属性

PointWithoutSlots 类定义（第 4~12 行）只定义了一个构造函数（第 7~12 行），这个构造函数初始化了点的 x 和 y 坐标。PointWithSlots 类（第 14~25 行）从 object 这个基类继承，并定义了一个 __slots__ 属性。它本质上是一个属性名列表，其中规定了类中允许包含的属性。一旦新类定义了 __slots__ 属性，这个类的对象便只能为名称在 __slots__ 列表中的属性赋值。如果客户试图为未包含在 __slots__ 中的属性赋值，Python 就会引发异常。

软件工程知识 9.10 如果新类定义了 __slots__ 属性，但类的构造函数没有初始化属性值，那么一旦创建该类的一个对象，Python 就会将 None 值指派给 __slots__ 中的每个属性。

软件工程知识 9.11 派生类会继承它的基类的 __slots__ 属性。然而，如果不希望程序为派生类的对象添加属性，派生类就必须定义自己的 __slots__ 属性。在派生类的 __slots__ 中，只包含允许的派生类属性名，但客户仍可为派生类的直接/间接基类所指定的属性赋值。

driver 程序（第 28~44 行）演示了定义 __slots__ 的类对象和没有定义 __slots__ 的类对象之间的差异。第 29~30 行分别创建 PointWithoutSlots 和 PointWithSlots 类的对象。第 32~44 行的 for 循环遍历每个对象，并试图将对象的 x 属性值替换成用户指定的值，这个值是在第 36 行获得的。第 41 行包含一个逻辑错误——程序的原意是修改对象的 x 属性的值，但错误地创建了一个属性，名为 X，并将用户输入的值指派给新属性。对于 PointWithoutSlots 类的对象（即 noSlots 对象），第 41 行会正常执行，不会引发异常，而且第 44 行会打印 x 属性原来的值。相反，对于 PointWithSlots 的对象（即 slots 对象），第 41 行会引发异常，因为对象的 __slots__ 属性中并不包含“X”这一名称。

图 9.16 的例子演示了为新类定义 __slots__ 属性的好处，即防止不慎创建错误的属性。使用新类有助于提升程序的性能，因为 Python 事先已经知道程序不会为一个对象添加新属性。因此，Python 能采取更高效的方式存储和操纵对象。__slots__ 的一个缺点在于，有经验的 Python 程序员偶尔需要动态添加对象属性，定义 __slots__ 会妨碍他们快速创建动态的、灵活的应用程序。

9.12.5 属性

Python 的新类可包含描述对象特性的“属性”（即 Properties，本书按国内传统，不区分 Attribute 和 Properties，一般都译成“属性”。在本节，“属性”是指 Properties；为与 Attribute 区分，将 Attribute 译

成特性——译者注)。程序使用对象-特性语法来访问一个对象的属性。但是,类定义可指定多达4个组件来创建一个属性,这4个组件分别是:一个get方法,在程序访问属性值时执行;一个set方法,在程序设置属性值时执行;一个delete方法,在程序删除值时执行(比如使用del关键字);以及一个文档化字符串,用于描述属性。get、set和delete方法所执行的任务可使对象数据保持一致性状态。所以,属性为程序员额外提供了一种途径来控制对对象数据的访问。

图9.17重新定义了Time类(该类以前用于演示属性访问),将hour、minute和second等作为属性包括进来。构造函数(第7~12行)创建私有特性__hour、__minute和__second。通常,使用了属性的类将它们特性定义成私有的,从而向类的客户隐藏数据。然后,类的客户可访问那个类的公共属性,再由属性对私有特性的值进行get和set操作。

```

1 # Fig. 9.17: TimeProperty.py
2 # Class Time with properties
3
4 class Time( object ):
5     """Class Time with hour, minute and second properties"""
6
7     def __init__( self, hourValue, minuteValue, secondValue ):
8         """Time constructor, takes hour, minute and second"""
9
10        self.__hour = hourValue
11        self.__minute = minuteValue
12        self.__second = secondValue
13
14    def __str__( self ):
15        """String representation of an object of class Time"""
16
17        return "%.2d:%.2d:%.2d" % \
18            ( self.__hour, self.__minute, self.__second )
19
20    def deleteValue( self ):
21        """Delete method for Time properties"""
22
23        raise TypeError, "Cannot delete attribute"
24
25    def setHour( self, value ):
26        """Set method for hour attribute"""
27
28        if 0 <= value < 24:
29            self.__hour = value
30        else:
31            raise ValueError, \
32                "hour (%d) must be in range 0-23, inclusive" % value
33
34    def getHour( self ):
35        """Get method for hour attribute"""
36
37        return self.__hour
38
39    # create hour property
40    hour = property( getHour, setHour, deleteValue, "hour" )
41
42    def setMinute( self, value ):
43        """Set method for minute attribute"""
44
45        if 0 <= value < 60:
46            self.__minute = value
47        else:
48            raise ValueError, \
49                "minute (%d) must be in range 0-59, inclusive" % value
50
51    def getMinute( self ):
52        """Get method for minute attribute"""
53
54        return self.__minute
55
56    # create minute property

```

```

57 minute = property( getMinute, setMinute, deleteValue, "minute" )
58
59 def setSecond( self, value ):
60     """Set method for second attribute"""
61
62     if 0 <= value < 60:
63         self.__second = value
64     else:
65         raise ValueError, \
66             "second (%d) must be in range 0-59, inclusive" % value
67
68 def getSecond( self ):
69     """Get method for second attribute"""
70
71     return self.__second
72
73 # create second property
74 second = property( getSecond, setSecond, deleteValue, "second" )

```

```

Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from TimeProperty import Time
>>>
>>> timel = Time( 5, 27, 19 )
>>> print timel
05:27:19
>>> print timel.hour, timel.minute, timel.second
5 27 19
>>>
>>> timel.hour, timel.minute, timel.second = 16, 1, 59
>>> print timel
16:01:59
>>>
>>> timel.hour = 25
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "TimeProperty.py", line 31, in setHour
    raise ValueError, \
ValueError: hour (25) must be in range 0-23, inclusive
>>>
>>> timel.minute = -3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "TimeProperty.py", line 48, in setMinute
    raise ValueError, \
ValueError: minute (-3) must be in range 0-59, inclusive
>>>
>>> timel.second = 99
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "TimeProperty.py", line 65, in setSecond
    raise ValueError, \
ValueError: second (99) must be in range 0-59, inclusive

```

图 9.17 属性——Time 类

deleteValue 方法（第 20~23 行）引发一个异常，防止客户删除一个特性。我们用这个方法创建客户不能删除的属性。每个属性（hour、minute 和 second）都定义了相应的 get 和 set 方法。每个 get 方法只取得对象引用作为自己的一个参数，并返回属性的值。每个 set 方法都取得两个参数（对象引用参数和属性的新值）。第 25~32 行为 hour 这一属性定义 set 方法（setHour）。如果新值在合适的范围内，方法就将新值指派给属性；否则，方法会引发一个异常。getHour 方法（第 34~37 行）是 hour 这一属性的 get 方法，它的作用很简单，就是返回相应的私有特性（__hour）的值。

第 40 行的内建函数 property 取得 4 个参数：一个 get 方法、一个 set 方法、一个 delete 方法以及一个文档化字符串，并返回类的一个属性。第 40 行为了创建 hour 属性，具体向 property 函数传递的是

getHour、setHour 和 deleteValue 方法，以及字符串"hour"。客户则使用点访问运算符（.）来访问属性。如果客户将一个属性作为“右值”使用，会执行属性的 get 方法。如果客户将一个属性作为“左值”使用，会执行属性的 set 方法。如果客户使用关键字 del 删除属性，则会执行属性的 delete 方法。类定义剩余的部分（第 42~74 行）定义了 minute（在第 57 行创建）以及 second（在第 74 行创建）这两个属性的 get 和 set 方法。

软件工程知识 9.12 property 函数不要求调用者必须传递所有这 4 个参数。相反，调用者可为关键字参数 fget、fset、fdel 和 doc 传递相应的值，分别指定属性的 get、set 和 delete 方法以及文档化字符串。

图 9.17 的交互式会话展示了属性的优势。类客户可通过点访问运算符访问一个对象的特性，但类的作者同时还能确保数据的完整性。相较于实现 __setattr__、__getattr__ 和 __delattr__ 等方法，属性具有更大的优势。例如，类的作者可明确声明客户能使用点访问运算符访问的特性。另外针对每个特性，类的作者都能编写单独的 get、set 和 delete 方法，不必用 if/else 逻辑判断具体要访问哪个特性。

本章讨论了继承机制，并解释了为什么继承有利于软件重用和数据抽象。本章讨论了继承的两个例子：一个关于结构化继承；另一个关于由抽象基类带头的类层次结构。我们还介绍了 Python 2.2 所支持的新的面向对象编程功能。为了继续对数据完整性的讨论，我们介绍了属性（Properties），它允许类的客户通过点访问运算符访问数据，同时允许类将私有数据保持一致性状态。数据隐藏和数据完整性是基本的面向对象软件设计原则。程序员要想用 Python 构建大型的、具有工业强度的软件系统，可通过本章和前两章的学习，为以后打下坚实的基础。

第 10 章 图形用户界面组件（一）

学习目标

- 理解图形用户界面设计原则
- 会用 Tkinter 模块构建图形用户界面
- 会创建和操纵标签、文本字段、按钮、复选框和单选钮
- 学习使用鼠标事件和键盘事件
- 理解和使用布局管理器

10.1 概述

用户可以使用“图形用户界面”(GUI)与程序进行交互。GUI(读作“GOO-ee”)使程序具有特别的“外观”和“感觉”。通过为程序提供一致的界面组件,稍有 GUI 程序常识的用户即可快速熟悉一个新程序。这减少了用户学习操纵程序所需的时间,并提高了他们使用程序的能力,从而有效改进了工作效率。

界面知识 10.1 风格一致的用户界面能使用户更快地掌握新的应用程序。

GUI 是用“GUI 组件”构建的。在 Python 中,将这些 GUI 组件称为“元件”,即 widget,它是 windows gadget(窗口配件)的缩写形式。GUI 组件是一种对象,用户可通过鼠标或键盘与它交互。图 10.1 展示了 GUI 例子,这是一个 Internet Explorer 窗口,图中标记了它的一些 GUI 组件。在“菜单栏”中包含一系列“菜单”,比如 File、Edit 和 View 等。菜单栏下方是一系列“按钮”,比如 Back、Search 和 History 等,每个按钮都能执行 Internet Explorer 所规定的一项任务。按钮下方是一个“文本字段”,用户可在其中输入一个网址。文本字段左侧是一个“标签”(即 Address),它指明文本字段的用途。菜单、按钮、文本字段和标签是 Internet Explorer GUI 的一部分。这些组件允许用户用鼠标单击一个元素,与 Internet Explorer 程序交互。

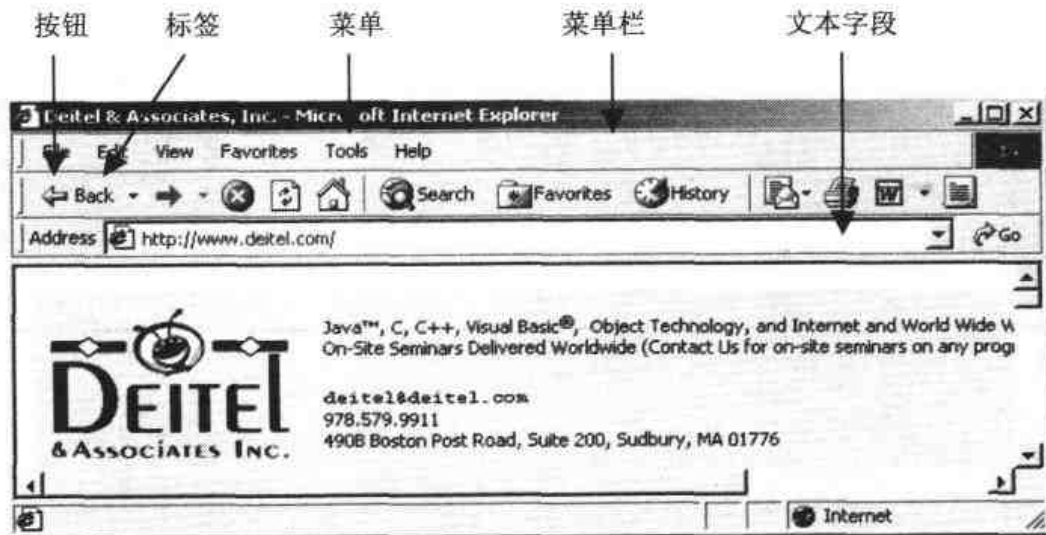


图 10.1 Internet Explorer 窗口中的 GUI 组件

Python 程序员可用“工具命令语言”(Tool Command Language, TCL)程序及其图形界面开发工具“工具包”(Tool Kit, Tk)来构建 GUI。要想了解这种脚本语言及其组件的详情,请访问 www.scriptics.com。

图 10.2 列出了 Tk 的一些常用 GUI 组件。本章和下一章要详细讨论它们以及其他 GUI 组件。

组件	说明
Frame	作为其他组件的容器使用
Label	显示不可编辑的文本或图标
Entry	接受用户的键盘输入, 或用于显示信息。是单行输入区域
Text	接受用户的键盘输入, 或用于显示信息。是多行输入区域
Button	单击按钮可触发一个事件
Checkbutton	一种选择组件, 只有选择和未选择两种状态
Radiobutton	一种选择组件, 只允许用户选择一个选项
Menu	显示一系列可供用户选择的选项
Canvas	显示文本、图像、线条或图形
Scale	允许用户使用滑杆选择一定范围内的整数值
Listbox	显示一个文本选项列表
Menubutton	显示弹出式菜单或下拉菜单
Scrollbar	为 Canvas、文本字段和列表显示一个滚动条

图 10.2 GUI 组件

10.2 Tkinter 简介

Python 中通常用 Tkinter 模块进行 GUI 编程, 因其是 Python 的标准 GUI 包, 与 Python 程序配套提供。^①虽然还有另一些 GUI 包可用于 Python, 但本书将以 Tkinter 为例。Tkinter 库为 Tk GUI 工具包提供了一个面向对象的接口。作为位于 Tk/TCL 顶部的一个面向对象的层, Tkinter 模块中的每个 Tk GUI 组件都是从 Widget 类继承的一个类 (如图 10.3 所示)。所有从 Widget 派生的类都具有公共的属性及行为。

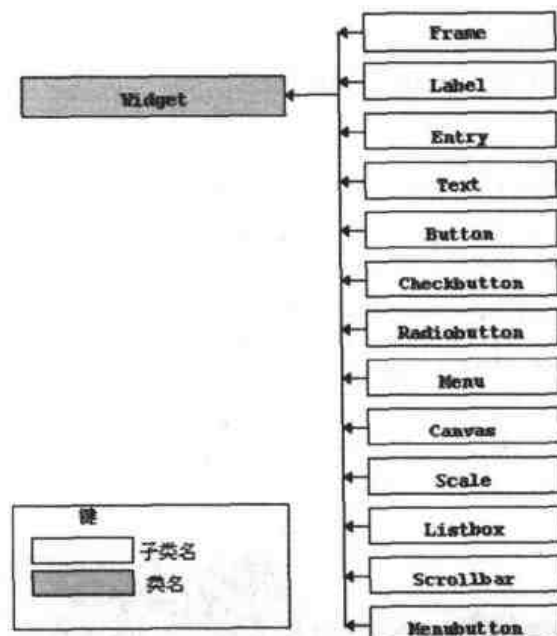


图 10.3 Widget 子类

^① Tkinter 模块可在多种平台之间移植。但是, 有的平台要求事先安装 Tcl/Tk 和 Tkinter。Deitel & Associates 公司网站 (www.deitel.com) 提供了针对不同平台的安装操作指南。

GUI 由一个顶级（或“父”）组件构成，该组件中可包含其他 GUI 组件。包含在“父”中的组件是顶级组件的“子”，每个“子”又可包含其他“子”。由于事先已经知道基类和派生类的关系，所以这种“父子”组件的概念不应该让您觉得陌生。程序为了在顶级组件的基础上构建一个 GUI，采取的方法是创建新组件，并将每个新组件都放在父组件中。

本章每个程序都通过从 Widget 的子类 Frame 继承，实现了一个 GUI。在我们的程序中，Frame 被作为顶级组件使用。以它为基础，将添加不同的“子”，以扩展 GUI 的功能。利用这种继承结构，可方便地在其他 GUI 程序中重用组件，同时有利于采取面向对象的编程方式。

移植性提示 10.1 Tkinter 模块可为 Unix、Macintosh 和 Windows 平台设计图形用户界面。

10.3 简单的 Tkinter 例子：Label 组件

标签用于显示文本或图像，以便在图形用户界面中提供指示或其他信息。图 10.4 演示的是 Label 类，它是用于表示一个标签组件的 Tkinter 类。

```

1 # Fig. 10.4: fig10_04.py
2 # Label demonstration.
3
4 from Tkinter import *
5
6 class LabelDemo( Frame ):
7     """Demonstrate Labels"""
8
9     def __init__( self ):
10         """Create three Labels and pack them"""
11
12         Frame.__init__( self ) # initializes Frame object
13
14         # frame fills all available space
15         self.pack( expand = YES, fill = BOTH )
16         self.master.title( "Labels" )
17
18         self.Label1 = Label( self, text = "Label with text" )
19
20         # resize frame to accommodate Label
21         self.Label1.pack()
22
23         self.Label2 = Label( self,
24                             text = "Labels with text and a bitmap" )
25
26         # insert Label against left side of frame
27         self.Label2.pack( side = LEFT )
28
29         # using default bitmap image as label
30         self.Label3 = Label( self, bitmap = "warning" )
31         self.Label3.pack( side = LEFT )
32
33 def main():
34     LabelDemo().mainloop() # starts event loop
35
36 if __name__ == "__main__":
37     main()

```

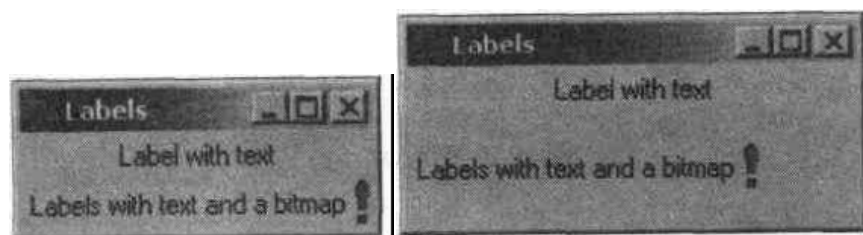


图 10.4 Label 演示

第 4 行导入 Tkinter 类定义以及预定义的值 (或称“常量”)。第 4 章介绍了怎样从一个模块导入所有元素:

```
from module import *
```

该语句可减少代码量, 因为特定的定义不需要通过模块名称来访问。然而, 如果不加思索地导入所有定义, 也可能导致错误。例如, 假定从定义了函数 len 的一个模块导入所有元素, 该函数的新定义就会覆盖 Python 原有的 len 定义。在这种情况下, 程序将无法判断一个序列的长度。所以, 为安全起见, 请只从那些明确声明可使用 import * 语句的模块 (比如 Tkinter) 导入全部元素。

LabelDemo 类 (第 6~31 行) 为我们的程序定义 GUI。该类从 Frame 类继承, 是 3 个 Label 组件的父容器。一旦客户创建一个 LabelDemo 对象, 同时执行类的 __init__ 方法 (第 9~31 行), 就会构建出完整的 GUI。第 12 行调用基类 Frame 的构造函数, 它会为整个应用程序创建一个顶级组件, 并初始化 Frame。

组件创建并初始化之后, 必须被放入其父容器中 (比如通过调用基类构造函数而创建的顶级组件)。pack 方法 (第 15 行) 及其关键字参数规定了组件应该如何放到它的“父”中, 以及应该放到什么地方。每个父组件都有特定大小的空间, 便于在其中放入子组件。另外, 每个子组件都有一个初始的默认大小。pack 方法一旦执行, 一个“布局管理器” (Layout Manager) 就会确定子组件的具体大小和位置, 它依据的是父容器中可用的空间。布局管理器的详情参见 10.10.1 节。

pack 方法的关键字参数影响了组件的大小。其中, 关键字参数 fill 指定组件应占据多大空间 (超出它的默认大小)。对于 fill, 可选值包括 X (所有可用的水平空间), Y (所有可用的垂直空间), BOTH (同时包括水平和垂直空间) 以及 NONE (默认值——不占任何额外空间)。所有子组件都放到其中仍有可用空间的“父”中之后, 那个“父”可能仍有可用的空间。关键字参数 expand 指定子组件是否应该占用父组件中任何多余的空间 (即未被其他组件占用的空间)。这个关键字的值可为 YES (扩充以占据多余空间) 或 NO (不进行扩充)。LabelDemo 对象占据由它的父 (顶级) 组件提供的所有可用空间, 因为 expand 和 fill 分别设为 YES 和 BOTH (第 15 行)。

界面知识 10.2 如果不设置选项, pack 方法会用它的默认设置将组件放到一个 GUI 中。如果程序员希望改变一个组件的位置, 可更改关键字参数。

良好编程习惯 10.1 使用 GUI 类之前, 请阅读 Python 联机文档, 学习类的方法和选项, 以理解其功能。

每个子组件都含有名为 master 的一个属性, 它引用这个“子”的父组件。第 16 行访问 LabelDemo 的父 (顶级) 组件, 并调用 title 方法将 GUI 的标题更改成“Labels”, 以便在 GUI 的标题栏上显示。

第 18 行创建一个标签对象。每个 GUI 组件的类构造函数所取得的第一个参数都对应于新对象的“父”。在这个例子中, self 是第一个参数, 它指明 Label 是 LabelDemo 组件的一个“子”。关键字参数 text 的值指明 Label 组件的内容。pack 方法 (第 21 行) 使用默认设置, 在 GUI 中插入 Label1。默认情况下, Label1 将占据窗口顶部的空间。

第 23~24 行创建第二个标签组件。第 27 行调用 pack 方法, 并为关键字参数 side 传递一个值, 该参数描述新组件的位置。值 LEFT 表明 Label2 要沿着窗口左侧显示。side 选项其他可能的值包括 BOTTOM、RIGHT 和 TOP (默认值)。这些选项还决定了当父容器的大小发生改变时, 子组件的位置和大小。图 10.4 显示了窗口大小发生变化后的结果。正如 side 选项所指定的那样, label1 将保持在容器顶部, 而 label2 和 label3 将停留在容器左侧。10.10.1 节讨论了 pack 方法的不同设置以及改变父容器的效果。

如果程序为关键字参数 bitmap 指定一个特定的值, 就可在标签中显示一幅图像。例如, 值“warning” (第 30 行) 会在 label3 上显示一幅警告位图。图 10.5 总结了可为 bitmap 选择的其他值。

除现成的位图图像之外, 程序员还可自行创建图像, 并用关键字参数 image 将其插入 GUI。注意, 在 image、bitmap 和 text 这几个关键字参数之间, 存在着优先顺序 (依次递减)。例如, 假定指定了 image 选项, 任何 bitmap 或 text 选项都会被忽略。类似地, 如果同时指定 bitmap 和 text 选项, text 选项就会被忽略。标签选项遵循一种优先级层次结构——最高优先级的选项的值会出现在 GUI 中, 其他则会被忽略。

具有最高优先级的标签是 image，其次是 bitmap，优先级最低的是 text。











位图图像名称	图像	位图图像名称	图像
error		hourglass	
gray75		info	
gray50		questhead	
gray25		question	
gray12		warning	

图 10.5 可选位图图像

第 3 个标签组件是 Label3，它的 side 选项设为 LEFT（第 31 行）。这个设置会使标签相对于 Label2 左对齐，而不是对齐 GUI 的边缘。10.10.1 节详细讲述了 pack 方法如何在 GUI 中排列组件。

第 33~37 行是通用于许多 GUI 程序的一项约定。第 36~37 行检测名称空间是否为“__main__”，并在条件成立（即为文件调用了解释器）的前提下调用 main 函数。如果文件作为一个模块导入，该条件就不成立。如果程序单独运行，而非作为模块在另一个程序中使用，main 函数就会执行。

main 函数创建一个 LabelDemo 对象，并调用它的主 loop 方法（第 34 行）。mainloop 方法启动 labelDemo GUI。该方法在必要时（比如在用户改变了 GUI 的大小之后）重画 GUI，并将事件发送给相应的组件（10.4 节会具体讨论事件）。一旦用户销毁（关闭）GUI，mainloop 方法就会终止。

10.4 事件处理模型

GUI 是“事件驱动”的；也就是说，一旦用户与 GUI 交互，GUI 组件就会生成“事件”（动作）。常见交互包括移动鼠标、单击一个鼠标按钮、在文本字段中输入、从菜单选择一个选项以及关闭一个窗口等等。发生用户交互时，一个事件会发送给程序。GUI 事件信息存储在 Event 类的一个对象中。对于一个事件驱动的程序来说，它是“异步”的——程序不知道事件会在何时发生。

为处理 GUI 事件，程序需要将事件与一个图形组件“绑定”，并实现一个“事件处理程序”（或称“回调”）。程序将一个事件同图形组件绑定（或关联）时，需指定要执行的动作。事件处理程序本身即是方法，它为响应一个关联的事件而被调用。

一个事件发生后，用户与之交互的 GUI 组件首先判断是否为该事件指定了一个事件处理程序。如果是，就执行与事件关联在一起的事件处理程序。例如，鼠标移经一个组件上方时，会发生“rollover”事件。而程序可能要求一个标签发生改变（例如更改标签背景色），以响应这个事件。在这种情况下，程序员可定义一个方法，它能改变标签的外观，并将 rollover 事件绑定到方法。这样，一旦鼠标移经标签，方法就会执行。

10.5 Entry 组件

Entry（输入）组件是一种特殊的屏幕区域，用户可在其中输入文本，程序员也可在其中显示一行文本。本节用一个程序演示了这种组件。一旦用户在 Entry 组件中输入文本，并按回车键，就会发生一个 <Return> 事件。如果为 Entry 组件绑定了一个事件处理程序，就会对该事件进行处理。在我们的例子中，<Return> 事件表明用户完成了文本输入。图 10.6 定义了 EntryDemo 类，它创建并操纵 4 个 Entry 文本字

段。一旦用户在活动字段中按下回车键, 程序就会显示字段中输入的文本。程序包含 2 个 Frame 对象, 这两个对象分别包含两个 Entry 组件。

```

1 # Fig. 10.6: fig10_06.py
2 # Entry components and event binding demonstration.
3
4 from Tkinter import *
5 from tkMessageBox import *
6
7 class EntryDemo( Frame ):
8     """Demonstrate Entrys and Event binding"""
9
10    def __init__( self ):
11        """Create, pack and bind events to four Entrys"""
12
13        Frame.__init__( self )
14        self.pack( expand = YES, fill = BOTH )
15        self.master.title( "Testing Entry Components" )
16        self.master.geometry( "325x100" ) # width x length
17
18        self.frame1 = Frame( self )
19        self.frame1.pack( pady = 5 )
20
21        self.text1 = Entry( self.frame1, name = "text1" )
22
23        # bind the Entry component to event
24        self.text1.bind( "<Return>", self.showContents )
25        self.text1.pack( side = LEFT, padx = 5 )
26
27        self.text2 = Entry( self.frame1, name = "text2" )
28
29        # insert text into Entry component text2
30        self.text2.insert( INSERT, "Enter text here" )
31        self.text2.bind( "<Return>", self.showContents )
32        self.text2.pack( side = LEFT, padx = 5 )
33
34        self.frame2 = Frame( self )
35        self.frame2.pack( pady = 5 )
36
37        self.text3 = Entry( self.frame2, name = "text3" )
38        self.text3.insert( INSERT, "Uneditable text field" )
39
40        # prohibit user from altering text in Entry component text3
41        self.text3.config( state = DISABLED )
42        self.text3.bind( "<Return>", self.showContents )
43        self.text3.pack( side = LEFT, padx = 5 )
44
45        # text in Entry component text4 appears as *
46        self.text4 = Entry( self.frame2, name = "text4",
47                            show = "*" )
48        self.text4.insert( INSERT, "Hidden text" )
49        self.text4.bind( "<Return>", self.showContents )
50        self.text4.pack( side = LEFT, padx = 5 )
51
52    def showContents( self, event ):
53        """Display the contents of the Entry"""
54
55        # acquire name of Entry component that generated event
56        theName = event.widget.winfo_name()
57
58        # acquire contents of Entry component that generated event
59        theContents = event.widget.get()
60        showinfo( "Message", theName + ": " + theContents )
61
62    def main():
63        EntryDemo().mainloop()
64
65    if __name__ == "__main__":
66        main()

```



图 10.6 Entry 组件和事件绑定演示

第 5 行从 `tkMessageBox` 模块导入类定义和常量。在这个模块中，包含了可显示对话框的函数，以便向用户显示消息。

`EntryDemo` 类的 `__init__` 方法调用基类构造函数，对 `EntryDemo` 进行 `pack` 操作，并指定程序标题（第 13~15 行）`geometry` 方法配置以像素为单位，指定顶级组件的长度和宽度（第 16 行）。第 18 行创建第一个 `Frame` 组件，即 `frame1`。`pack` 方法调用（第 19 行）引入了另一个选项，即 `pady`，它指定了 `frame1` 和父容器中的其他 GUI 组件之间的垂直空白间距。类似地，`padx`（稍后会在程序中使用）指定了组件之间的水平空白间距。

第 21 行创建 `Entry` 组件 `text1`。其中，选项 `name` 为 `Entry` 分配一个名称。有了名称之后，事件处理程序就能用那个名称标识可能在其中发生事件的组件。

界面知识 10.3 如果程序员不指定名称，Tkinter 就会为每个组件分配一个不重复的名称。要想获得一个组件的完整名称，将组件对象传给 `str` 函数即可。

`bind` 方法（第 24 行）将一个 `<Return>` 事件同 `text1` 组件关联在一起。一旦用户按下回车键，`<Return>` 事件就会发生。`bind` 方法要取得两个参数：第一个参数是事件类型（事件格式），第二个参数是要与事件绑定的那个方法的名称。在本例中，一旦在 `text1` 中发生 `<Return>` 事件，就会执行 `showContents` 方法。

第 30~32 行创建 `Entry` 组件 `text2`，并对其进行 `pack` 操作。`insert` 方法在 `Entry` 组件中写入文本（第

30 行), 它要取得两个参数: 文本插入的位置和包含待插入文本的一个字符串。如果第一个参数的值是 INSERT, 文本会在当前光标位置插入。另外, 文本也可在 Entry 组件的末尾插入。例如以下调用:

```
insert( END, text )
```

会将 text 附加到组件中已经显示的文本的末尾。

还可使用 delete 方法从 Entry 组件中删除文本。例如以下调用:

```
delete( start, finish )
```

可删除 Entry 组件中 start 到 finish 之间的所有文本。如果将第二个参数设为 END, 上述方法会一直删除到文本字段的末尾。在 Entry 组件中, 第一个位置的编号是 0; 因此, delete(0, END)实际会从 Entry 组件中删除全部文本。

第 34~35 行创建第二个 Frame 组件, 即 frame2, 并对其进行 pack 操作。程序将使一个 Frame 位于另一个 Frame 的下方, 从而在屏幕上显示了两行, 并在每一行插入两个 Entry 组件。具体地说, 程序是将 text1 和 text2 这两个 Entry 组件插入 frame1, 将 text3 和 text4 插入 frame2。

第 41~43 行采用与前两个 Entry 组件一样的方式创建 text3, 并对其进行 pack 操作。组件同<Return>事件绑定到一起 (第 42 行)。这个例子演示了如何用 config 方法禁用 text3。config 方法允许用户指定一对“关键字-值”, 从而对组件进行配置 (第 41 行)。将 state 选项设为 DISABLED, 即可禁用 Entry 组件。这样, 用户将无法编辑其中的文本。结果就是, text3 将无法生成一个<Return>事件。如果希望显示文本, 但不希望用户对其进行编辑, 就可考虑禁用 Entry 组件。

第 46~50 行采用相同的方式创建第 4 个 Entry 组件, 即 text4, 并对其进行 pack 操作。该组件允许用户输入的机密信息。show 选项规定要在文本框中显示的字符, 并用它替代用户输入的所有文本 (第 47 行)。在本例中, 星号 (*) 将替代程序插入的默认文本“Hidden text”。此外, 用户在 Entry 组件中输入文本时, 所有文本都会自动替换成星号。

showContents 方法 (第 52~60 行) 是各个 Entry 组件中生成的全部<Return>事件的事件处理程序。在 Python 中, 大多数事件处理程序都要取得对 Event 对象的一个引用作为参数; 而 Event 对象可以设置多个属性。生成事件的组件可通过对象的 widget 属性 (即 event.widget) 而获得。在我们的程序中, event.widget 引用 4 个 Entry 组件之一 (其<Return>事件已经和 showContents 方法绑定)。

常见编程错误 10.1 针对特定 GUI 组件, 如果没有为一种事件类型绑定相应的事件处理程序, 那么即使发生该事件, 它也不会得到处理。

Widget 的 wininfo_name 方法 (第 56 行) 返回组件名称。Entry 的 get 方法 (第 59 行) 返回 Entry 的内容。事件处理程序使用这两个返回值来生成一条消息, 并向用户显示。tkMessageBox 的 showinfo 函数 (第 60 行) 显示一个标记为“Message”的对话框, 其中包含了生成事件的那个 Entry 组件的名称与内容。图 10.6 末尾的屏幕截图演示了每个 Entry 组件接收到<Return>事件后发生的事情。

10.6 Button 组件

Button (按钮) 这个 GUI 组件会在被按下时生成一个事件。按钮简化了程序操作, 用户可选择合适的按钮来执行一个希望的动作, 而不必人工输入命令。按钮是用 Button 类创建的, 该类继承自 Widget。Button 上显示的文本或图像称为“按钮标签”。GUI 中可显示多个按钮, 但通常每个按钮都应该有惟一的按钮标签。

界面知识 10.4 使用多个具有相同标签的按钮会产生歧义。请分别为每个按钮提供不重复的标签。

图 10.7 创建了两个按钮, 并证明 Button 组件可像 Label 组件那样显示文本或图像。

第18~19行创建一个Button,名为plainButton。text选项用于设置按钮的标签。关键字参数command指定了在使用户选择按钮之后要执行的事件处理程序。在我们的例子中,plainButton的标签是"Plain Button",它的事件处理程序是pressedPlain方法。

第20~21行将rolloverEnter和rolloverLeave方法分别绑定到plainButton按钮的<Enter>和<Leave>事件。其中,<Enter>事件会在用户将鼠标指针移到按钮上方时发生,<Leave>事件会在鼠标指针离开按钮时发生。10.8节详细讨论了鼠标事件的问题。

包括Button在内的许多Tkinter组件都能通过为其构造函数或者config方法指定image参数的方式来显示图像。要显示的图像必须是能载入一个图像文件的Tkinter类的一个对象。PhotoImage便符合这一要求,它支持3种图像格式,即GIF、JPEG和PGM。针对这些类型,相应的文件扩展名是.gif、.jpg(或.jpeg)以及.pgm(或.ppm)。另外,BitmapImage类也符合这一要求,它支持位图(BMP)图像格式(.bmp)。第25行创建了一个PhotoImage对象。logotiny.gif文件包含要载入和存储到PhotoImage对象中的图像(该文件位于和程序相同的目录中)。程序将新建的PhotoImage对象指派给myImage引用。

```

1 # Fig. 10.7: fig10_07.py
2 # Button demonstration.
3
4 from Tkinter import *
5 from tkMessageBox import *
6
7 class PlainAndFancy( Frame ):
8     """Create one plain and one fancy button"""
9
10    def __init__( self ):
11        """Create two buttons, pack them and bind events"""
12
13        Frame.__init__( self )
14        self.pack( expand = YES, fill = BOTH )
15        self.master.title( "Buttons" )
16
17        # create button with text
18        self.plainButton = Button( self, text = "Plain Button",
19                                command = self.pressedPlain )
20        self.plainButton.bind( "<Enter>", self.rolloverEnter )
21        self.plainButton.bind( "<Leave>", self.rolloverLeave )
22        self.plainButton.pack( side = LEFT, padx = 5, pady = 5 )
23
24        # create button with image
25        self.myImage = PhotoImage( file = "logotiny.gif" )
26        self.fancyButton = Button( self, image = self.myImage,
27                                command = self.pressedFancy )
28        self.fancyButton.bind( "<Enter>", self.rolloverEnter )
29        self.fancyButton.bind( "<Leave>", self.rolloverLeave )
30        self.fancyButton.pack( side = LEFT, padx = 5, pady = 5 )
31
32    def pressedPlain( self ):
33        showinfo( "Message", "You pressed: Plain Button" )
34
35    def pressedFancy( self ):
36        showinfo( "Message", "You pressed: Fancy Button" )
37
38    def rolloverEnter( self, event ):
39        event.widget.config( relief = GROOVE )
40
41    def rolloverLeave( self, event ):
42        event.widget.config( relief = RAISED )
43
44    def main():
45        PlainAndFancy().mainloop()
46
47    if __name__ == "__main__":
48        main()

```

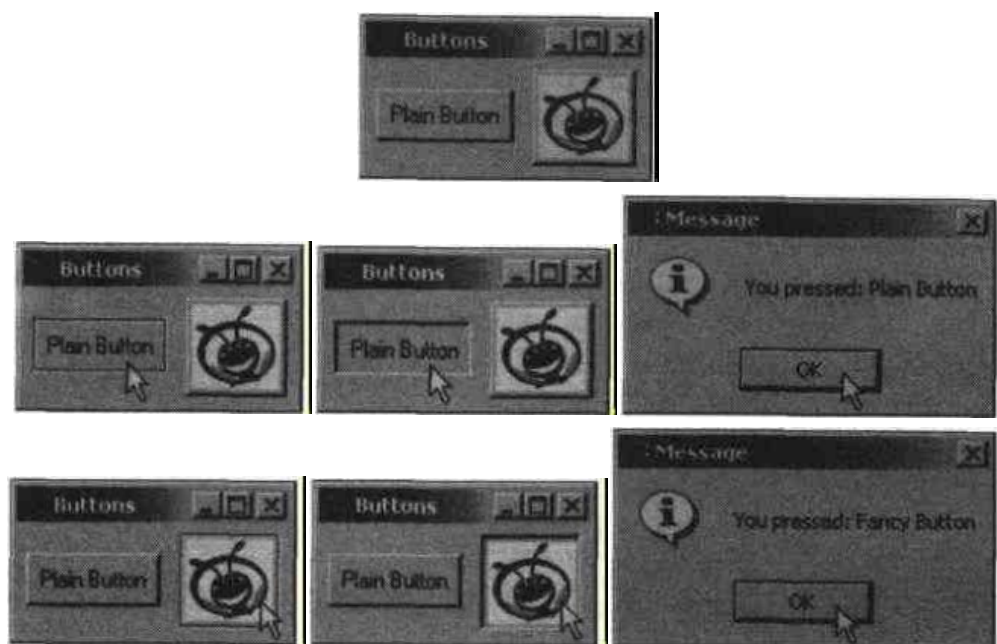


图 10.7 Button 演示

第 26~27 行创建 fancyButton, 并将 image 属性设为 myImage。和 Label 组件一样, image 属性的优先级比 text 和 bitmap 属性更高。所以如果指定了 text 或 bitmap 属性, 它们会被忽略。

fancyButton 的事件处理程序是 pressedFancy。注意, 方法 pressedPlain(第 32~33 行)和 pressedFancy(第 35~36 行)没有取得一个 Event 对象作为自己的参数。这是因为 Button 的事件处理程序(回调方法)不要求用 Event 对象作为参数。但是, 没有 Event 对象, 事件处理程序无法判断事件是在哪个组件中发生的, 所以必须为每个 Button 都单独指定一个事件处理程序, 以标识出正确组件。pressedPlain 和 pressedFancy 方法会创建“Message”对话框, 它向用户指出具体是什么按钮生成了事件。

良好编程习惯 10.2 为每个 Button 都单独定义一个回调方法, 这有助于避免混淆, 确保实现目标行为, 并简化 GUI 的调试。

rolloverEnter 方法(第 38~39 行)和 rolloverLeave 方法(第 41~42 行)为相应的事件创建一种“rollover”效果, 以改变组件的显示效果。两个方法改变的都是组件的“relief”(高低), 即组件相对于周围的组件是凹陷还是凸起。其中, rolloverEnter 方法将组件的 relief 选项设为 GROOVE(凹陷); rolloverLeave 则将 relief 设为 RAISED(凸起)。

界面知识 10.5 通过改变按钮的显示效果, 可提供一种视觉线索, 让用户知道自己已经选中了按钮, 而且发生了相应的动作。

10.7 Checkbutton 和 Radiobutton 组件

Tkinter 定义了 Checkbutton(复选钮)和 Radiobutton(单选钮)这两个 GUI 组件, 它们具有 on/off(开/关)或者 true/false(真/假)值。Checkbutton 和 Radiobutton 是 Widget 的子类。尽管取值相同, 但 Checkbutton 和 Radiobutton 适用于不同的环境。下面首先讨论 Checkbutton 类。

平常所说的“复选框”是一种小的白色方块, 它要么空白, 要么包含一个勾号。选定一个复选框后, 会在框中出现一个黑色勾号。复选框的用法则没有任何限制, 完全可以同时选定任意数量的复选框。复选框旁边显示的文本称为“复选框标签”。Tkinter 的 Checkbutton 组件实际就是复选框组件。

图 10.8 使用两个 Checkbutton 对象修改 Entry 组件中所示文本的字形。一经选中, 一个 Checkbutton

会为文本应用加粗字形，另一个则应用倾斜字形。Checkbutton 最初是没有选定的。

```

1  # Fig. 10.8: fig10_08.py
2  # Checkbuttons demonstration.
3
4  from Tkinter import *
5
6  class CheckFont( Frame ):
7      """An area of text with Checkbutton controlled font"""
8
9      def __init__( self ):
10         """Create an Entry and two Checkbuttons"""
11
12         Frame.__init__( self )
13         self.pack( expand = YES, fill = BOTH )
14         self.master.title( "Checkbutton Demo" )
15
16         self.frame1 = Frame( self )
17         self.frame1.pack()
18
19         self.text = Entry( self.frame1, width = 40,
20                             font = "Arial 10" )
21         self.text.insert( INSERT, "Watch the font style change" )
22         self.text.pack( padx = 5, pady = 5 )
23
24         self.frame2 = Frame( self )
25         self.frame2.pack()
26
27         # create boolean variable
28         self.boldOn = BooleanVar()
29
30         # create "Bold" checkbutton
31         self.checkBold = Checkbutton( self.frame2, text = "Bold",
32                                       variable = self.boldOn, command = self.changeFont )
33         self.checkBold.pack( side = LEFT, padx = 5, pady = 5 )
34
35         # create boolean variable
36         self.italicOn = BooleanVar()
37
38         # create "italic" checkbutton
39         self.checkItalic = Checkbutton( self.frame2,
40                                         text = "Italic", variable = self.italicOn,
41                                         command = self.changeFont )
42         self.checkItalic.pack( side = LEFT, padx = 5, pady = 5 )
43
44         def changeFont( self ):
45             """Change the font based on selected Checkbuttons"""
46
47             desiredFont = "Arial 10"
48
49             if self.boldOn.get():
50                 desiredFont += " bold"
51
52             if self.italicOn.get():
53                 desiredFont += " italic"
54
55             self.text.config( font = desiredFont )
56
57         def main():
58             CheckFont().mainloop()
59
60         if __name__ == "__main__":
61             main()

```

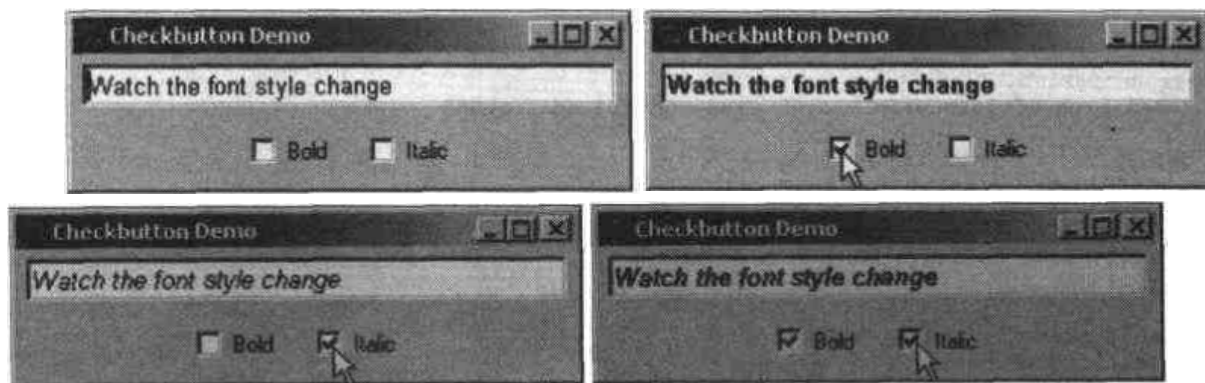


图 10.8 用 Checkbutton 设置字形

第 19~20 行创建名为 `text` 的一个 `Entry` 组件。第 21 行在其中插入文本“Watch the font style change”。这些文本会被应用“Arial 10”这一字形。注意 `font` 属性指定了 `Entry` 组件所用的字体。表示字体的一个办法是使用包含字体名称、字号和字形的一个字符串。既可以根本不指定字形，也可以指定多种字形。以下网址的联机文档“Introduction to Tkinter”介绍了可用的字体以及字形：

www.pythonware.com/library/Tkinter/introduction/x444fonts.htm

`boldOn` (第 28 行) 和 `italicOn` (第 36 行) 都是 `BooleanVar` 的一个对象，它们其实是 Tkinter 整数变量，值要么为 0，要么为 1。variable 选项要求取得 Tkinter 的 `Variable` 类的一个对象。`BooleanVar` 正是从 Tkinter 的 `Variable` 类继承的。`Variable` 类相当于所有 Python 变量的一个容器。各个 Tkinter 类用 `Variable` 对象维护与一个特定组件有关的信息。举个例子来说，`CheckButton` 类用一个 `BooleanVar` 对象存储按钮的“状态”（勾选或未勾选）。我们的程序创建 `BooleanVar` 引用，并将其传给 `CheckButton` 构造函数。这样一来，事件处理程序就可判断出用户是否勾选了任何复选框。

第 31~32 行创建了一个 `Checkbutton` 组件，名为 `checkBold` 的。`text` 选项指定要在复选框旁边显示“Bold”（加粗）字样，从而描述该复选框的作用。`Checkbutton` 组件的 `command` 属性指定用户勾选或撤选复选框时要执行的事件处理程序。在这种情况下，我们将 `changeFont` 方法设为事件处理程序。组件的 `variable` 选项传递组件用于维护其状态信息的 `BooleanVar` 对象。一旦用户单击复选框，就会发生两件事情：它的 `BooleanVar` 值从 0 变成 1，或从 1 变成 0 以及执行事件处理程序 `changeFont`。第 38~40 行创建 `checkItalic`，这个 `CheckButton` 对象的行为与 `checkBold` 相似。

`changeFont` 方法（第 44~55 行）将字符串 `desiredFont` 初始化为原始的“Arial 10”字体。`BooleanVar` 的 `get` 方法返回变量的值。如果用户勾选 `checkBold` 复选框，程序会为 `desiredFont` 附加“bold”字符串（第 50 行）。类似地，如果用户勾选 `checkItalic` 复选框，程序会为 `desiredFont` 附加“italic”字符串（第 53 行）。注意，每个字符串都以一个空格开头，从而确保在为字体附加了一种字形后，会包含一个空格（例如“Arial 10 italic”）。然后，方法调用 `config` 方法，将 `text` 组件的字体变成 `desiredFont`。

使用 `Radiobutton` 类可创建单选钮，单选钮的行为类似于复选框，同样只有两种状态：选择和撤选。但与复选框不同的是，单选钮表示的是一系列“互斥”选项。在一组单选钮中，一次只能选一个。如果选择一组中的一个单选钮，其他所有单选钮都会被自动撤选。

界面知识 10.6 如果只允许用户从一组选项中选择一个，请使用单选钮。

界面知识 10.7 如果允许用户从一组选项中选择多个，请使用复选框。

图 10.9 的程序类似图 10.8，都允许用户更改一个 `Entry` 组件中的文本的字形。但是，由于使用的是单选钮，所以这个例子每次只允许从一组字形中选择一种。

```
1 # Fig. 10.9: fig10_09.py
2 # Radiobuttons demonstration.
3
```

```
4 from Tkinter import *
5
6 class RadioFont( Frame ):
7     """An area of text with Radiobutton controlled font"""
8
9     def __init__( self ):
10         """Create an Entry and four Radiobuttons"""
11
12         Frame.__init__( self )
13         self.pack( expand = YES, fill = BOTH )
14         self.master.title( "Radiobutton Demo" )
15
16         self.frame1 = Frame( self )
17         self.frame1.pack()
18
19         self.text = Entry( self.frame1, width = 40,
20                             font = "Arial 10" )
21         self.text.insert( INSERT, "Watch the font style change" )
22         self.text.pack( padx = 5, pady = 5 )
23
24         self.frame2 = Frame( self )
25         self.frame2.pack()
26
27         fontSelections = [ "Plain", "Bold", "Italic",
28                             "Bold/Italic" ]
29         self.chosenFont = StringVar()
30
31         # initial selection
32         self.chosenFont.set( fontSelections[ 0 ] )
33
34         # create group of Radiobutton components with same variable
35         for style in fontSelections:
36             aButton = Radiobutton( self.frame2, text = style,
37                                     variable = self.chosenFont, value = style,
38                                     command = self.changeFont )
39             aButton.pack( side = LEFT, padx = 5, pady = 5 )
40
41         def changeFont( self ):
42             """Change the font based on selected Radiobutton"""
43
44             desiredFont = "Arial 10"
45
46             if self.chosenFont.get() == "Bold":
47                 desiredFont += " bold"
48             elif self.chosenFont.get() == "Italic":
49                 desiredFont += " italic"
50             elif self.chosenFont.get() == "Bold/Italic":
51                 desiredFont += " bold italic"
52
53             self.text.config( font = desiredFont )
54
55         def main():
56             RadioFont().mainloop()
57
58         if __name__ == "__main__":
59             main()
```



图 10.9 用 Radiobutton 设置字形

fontSelections 序列 (第 27~28 行) 列出了程序需要的几种字形。第 29~32 行定义一个 StringVar 对象, 名为 chosenFont, 并将它的初始值设为默认字形, 即 "Plain"。和 BooleanVar 类似, StringVar 也是 Tkinter 的 Variable 类的一个子类, 它作为字符串变量的一个容器使用。复选框的例子用一个 BooleanVar 跟踪复选框的状态。与此不同的是, 在图 10.9 中, 一组单选钮要用一个 StringVar 来存储当前选定的单选钮的值 (即它的名称)。这一组单选钮修改的是同一个 Variable 对象。如果同时定义了几组单选钮, 程序员必须为每个组都分配一个 Variable 对象。一旦用户选择 (单击) 某个特定的单选钮, 所选的单选钮会修改指定的 Variable 对象, 并执行相应的事件处理程序。我们的事件处理程序 (changeFont) 会获取 StringVar 对象, 确定具体选择的是哪一个按钮。

第 35~39 行为 fontSelections 列表中的每个字形 ("Plain", "Bold", "Italic" 和 "Bold/Italic") 都创建一个 Radiobutton 组件, 并对其进行 pack 操作。for 循环为每个按钮的 text 和 value 选项分配一个字形。text 选项指定要在单选钮旁显示的文本, 而 value 指定按钮名称。variable 属性将 chosenFont 这个 StringVar 对象与每一个 Radiobutton 组件关联起来, command 选项将 changeFont 方法注册为每个按钮的事件处理程序。一旦用户单击某个单选钮, 包含在 StringVar 对象中的字符串就会发生改变, 以包含该按钮的值, 并执行 changeFont 方法。

changeFont 方法 (第 41~53 行) 将 desiredFont 字符串初始化成 "Arial 10"。如果选中一个单选钮, changeFont 会将目标字形附加到 desiredFont。get 方法获得 chosenFont 的当前值。在这个例子中, changeFont 用一个 if/elif 结构强调每次只能选中一个单选钮 (使用相同的变量)。

10.8 鼠标事件处理

本节解释如何处理 "鼠标事件"。用户与鼠标交互时, 就会发生这样的事件。图 10.10 总结了几种常见鼠标事件, 图 10.11 则演示了如何在一个 GUI 程序中对其进行处理。所有 Tkinter 事件都用字符串描述, 格式为 <modifier-type-detail>。其中, type (比如 Button 和 Return) 指定事件种类, modifier 是指 Double 这样的前缀, detail 则是指具体的鼠标按钮。

事件格式	说明
<ButtonPress- <i>n</i> >	鼠标指针在组件上方时, 鼠标按钮 <i>n</i> 被选择 (按下)。 <i>n</i> 可以是 1 (左键), 2 (中键) 或者 3 (右键), 例如 <ButtonPress-1>
<Button- <i>n</i> >, < <i>n</i> >	都是 <ButtonPress- <i>n</i> > 的一种简化形式
<ButtonRelease- <i>n</i> >	鼠标按钮 <i>n</i> 被松开
<B <i>n</i> -Motion>	在按住按钮 <i>n</i> 的同时, 鼠标发生移动
<Prefix-Button- <i>n</i> >	对组件双击或三击了鼠标按钮 <i>n</i> 。Prefix 可以选择 Double (双击) 或 Triple (三击)
<Enter>	鼠标指针进入组件
<Leave>	鼠标指针离开组件

图 10.10 鼠标事件格式

```

1 # Fig. 10.11: fig10_11.py
2 # Mouse events example.
3
4 from Tkinter import *
5
6 class MouseLocation( Frame ):
7     """Demonstrate binding mouse events"""
8
9     def __init__( self ):
10         """Create a Label, pack it and bind mouse events"""
11
12         Frame.__init__( self )
13         self.pack( expand = YES, fill = BOTH )
14         self.master.title( "Demonstrating Mouse Events" )
15         self.master.geometry( "275x100" )
16
17         self.mousePosition = StringVar() # displays mouse position
18         self.mousePosition.set( "Mouse outside window" )
19         self.positionLabel = Label( self,
20             textvariable = self.mousePosition )
21         self.positionLabel.pack( side = BOTTOM )
22
23         # bind mouse events to window
24         self.bind( "<Button-1>", self.buttonPressed )
25         self.bind( "<ButtonRelease-1>", self.buttonReleased )
26         self.bind( "<Enter>", self.enteredWindow )
27         self.bind( "<Leave>", self.exitedWindow )
28         self.bind( "<B1-Motion>", self.mouseDragged )
29
30     def buttonPressed( self, event ):
31         """Display coordinates of button press"""
32
33         self.mousePosition.set( "Pressed at { " + str( event.x ) +
34             ", " + str( event.y ) + " }" )
35
36     def buttonReleased( self, event ):
37         """Display coordinates of button release"""
38
39         self.mousePosition.set( "Released at { " + str( event.x ) +
40             ", " + str( event.y ) + " }" )
41
42     def enteredWindow( self, event ):
43         """Display message that mouse has entered window"""
44
45         self.mousePosition.set( "Mouse in window" )
46
47     def exitedWindow( self, event ):
48         """Display message that mouse has left window"""
49
50         self.mousePosition.set( "Mouse outside window" )
51
52     def mouseDragged( self, event ):
53         """Display coordinates of mouse being moved"""
54
55         self.mousePosition.set( "Dragged at [ " + str( event.x ) +
56             ", " + str( event.y ) + " ]" )
57
58 def main():
59     MouseLocation().mainloop()
60
61 if __name__ == "__main__":
62     main()

```

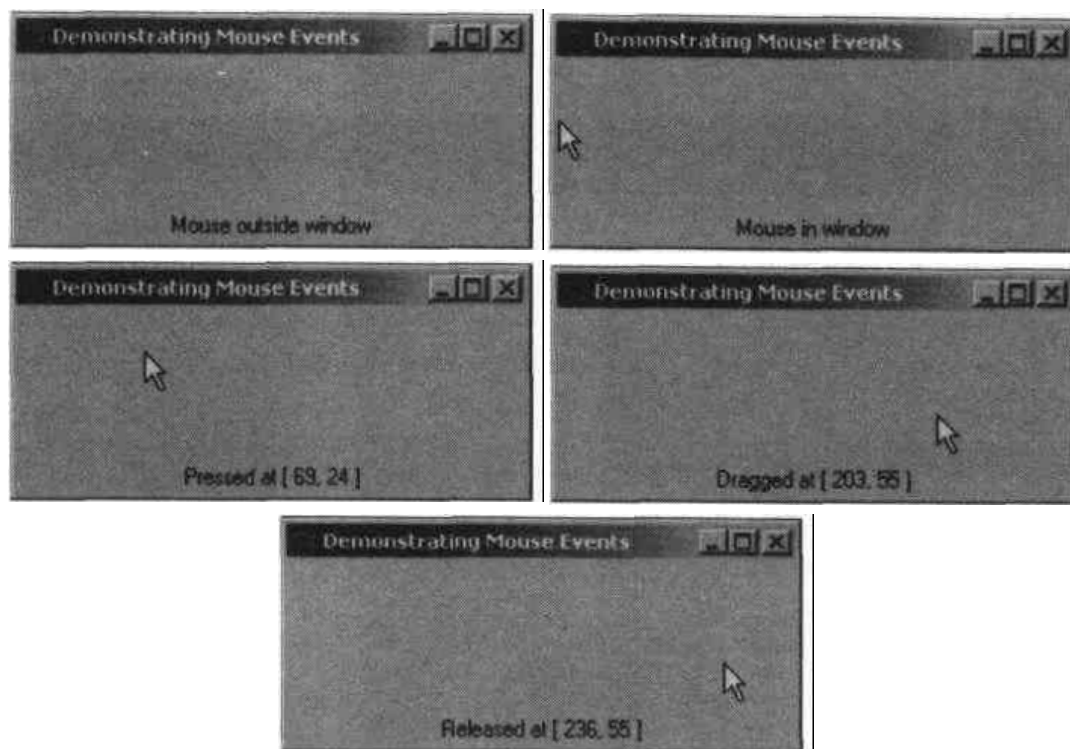


图 10.11 鼠标事件演示

第 17~18 行创建了一个 `StringVar` 对象, 名为 `mousePosition`, 并把它值初始化成 "Mouse outside window" (鼠标在窗口之外)。第 19~21 行创建并 `pack` 名为 `positionLabel` 的一个标签组件, 将 `textvariable` 选项设为 `mousePosition`。`textvariable` 选项将标签组件显示的文本与一个 `StringVar` 对象关联起来。该选项必须与一个 `Tkinter Variable` 对象关联 (注意, 图 10.4 演示了标签组件的 `text` 选项, 它与一个 Python 变量关联)。一旦对象 (目前是 `mousePosition`) 的字符串值发生变化, `positionLabel` 就会随之更新。

第 24~28 行将几个常用的鼠标事件绑定到窗口。鼠标指针位于窗口内部时, 一旦鼠标左键按下或松开, 就会产生一个事件。另外, 当鼠标指针进入或离开窗口, 或在按住左键不放的时候移动鼠标, 都会产生相应的事件。

生成 `<Button-1>` 或 `<ButtonRelease-1>` 事件后, 会分别执行 `buttonPressed` 方法 (第 30~34 行) 和 `buttonReleased` 方法 (第 36~40 行)。它们会调用 `set` 方法, 从而更改变量 `mousePosition` 的值, 提醒用户发生的事件。在鼠标事件的 `Event` 对象中, 包含了用于描述事件发生位置的 `x` 坐标和 `y` 坐标 (分别存储于 `Event` 对象的 `x` 和 `y` 属性中)。

一旦鼠标指针进入程序窗口, 就会执行 `enterWindow` 方法 (第 42~45 行)。鼠标指针离开窗口时, 则会执行 `exitedWindow` 方法 (第 47~50 行)。如屏幕截图所示, 每个方法都会打印一条合适的消息, 指出鼠标是否在 `MouseLocation` 对象上方。这些方法会修改 `mousePosition` 这个 `StringVar` 对象中的值, 以更新标签中显示的文本。

在不不同于事件处理程序 `buttonPressed` 和 `buttonReleased` 的其他情况下, 有可能触发事件处理程序 `mouseDragged` (第 52~56 行)。要想触发一个 `<B1-Motion>` 事件, 必须符合两个条件: 按钮 `B1` 必须按下, 而且鼠标必须移动。如满足这些条件, `<B1-Motion>` 事件就会按照操作系统所定义的频率发生。换言之, 在一个操作系统中, 朝右边拖动鼠标可能触发 50 个 `<B1-Motion>` 事件, 但在另一个不同的操作系统中, 这个频率可能会低一些。对于每个 `<B1-Motion>` 事件, `mouseDragged` 方法都会显示出初始事件和坐标。

鼠标可能是双键的, 也可能是三键的。取决于用户按下的鼠标按钮, 程序可能需要采取不同的动作。图 10.12 的程序演示了如何区分不同的鼠标按钮。

```

1 # Fig. 10.12: fig10_12.py
2 # Mouse button differentiation.
3
4 from Tkinter import *
5
6 class MouseDetails( Frame ):
7     """Demonstrate mouse events for different buttons"""
8
9     def __init__( self ):
10         """Create a Label, pack it and bind mouse events"""
11
12         Frame.__init__( self )
13         self.pack( expand = YES, fill = BOTH )
14         self.master.title( "Mouse clicks and buttons" )
15         self.master.geometry( "350x150" )
16
17         # create mousePosition variable
18         self.mousePosition = StringVar()
19         positionLabel = Label( self,
20             textvariable = self.mousePosition )
21         self.mousePosition.set( "Mouse not clicked" )
22         positionLabel.pack( side = BOTTOM )
23
24         # bind event handler to events for each mouse button
25         self.bind( "<Button-1>", self.leftClick )
26         self.bind( "<Button-2>", self.centerClick )
27         self.bind( "<Button-3>", self.rightClick )
28
29     def leftClick( self, event ):
30         """Display coordinates and indicate left button clicked"""
31
32         self.showPosition( event.x, event.y )
33         self.master.title( "Clicked with left mouse button" )
34
35     def centerClick( self, event ):
36         """Display coordinates and indicate center button used"""
37
38         self.showPosition( event.x, event.y )
39         self.master.title( "Clicked with center mouse button" )
40
41     def rightClick( self, event ):
42         """Display coordinates and indicate right button clicked"""
43
44         self.showPosition( event.x, event.y )
45         self.master.title( "Clicked with right mouse button" )
46
47     def showPosition( self, x, y ):
48         """Display coordinates of button press"""
49
50         self.mousePosition.set( "Pressed at : " + str( x ) + ", " +
51             str( y ) + " : " )
52
53     def main():
54         MouseDetails().main_loop()
55
56 if __name__ == "__main__":
57     main()

```

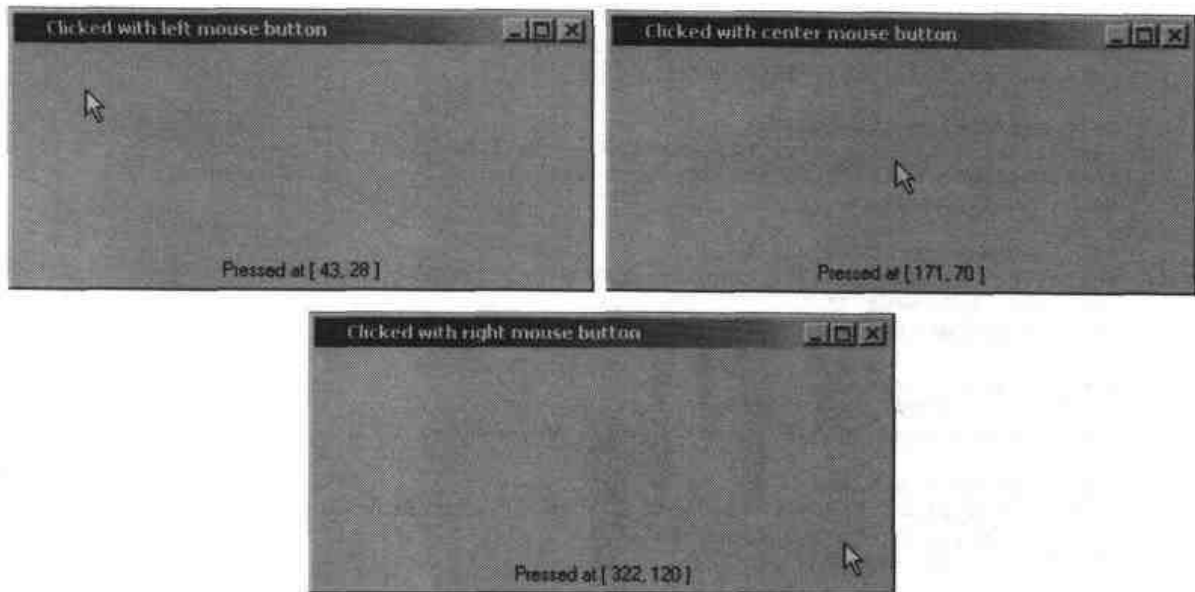


图 10.12 区分鼠标按钮

图 10.12 类似于图 10.11, 区别在于前者中的第 25~27 行通过更改事件格式 (`<Button-n>`) 中的编号, 将不同鼠标按钮的方法与事件绑定在一起。鼠标指针在窗口内部时, 如果用户按下一个按钮, 窗口标题就会发生变化, 指出具体按下的是哪一个按钮。每个事件处理程序都会调用 `showPosition` 方法 (第 47~51 行), 该方法显示了鼠标事件的坐标。

10.9 键盘事件处理

本节解释如何为“键盘事件”绑定事件处理程序。按下和松开键盘按键时, 就会发生这些事件。图 10.13 总结了键盘事件的所有格式。图 10.14 演示了如何为键盘事件绑定方法。为避免混淆, 我们不使用 `<KeyPress>` 和 `<KeyPress-key>` 事件的简化形式。

事件格式	事件说明
<code><KeyPress></code>	按下任意键
<code><KeyRelease></code>	松开任意键
<code><KeyPress-key></code> , <code><KeyRelease-key></code>	按下或松开 <code>key</code>
<code><Key></code> , <code><Key-key></code>	<code><KeyPress></code> 和 <code><KeyPress-key></code> 的简化形式
<code><key></code>	<code><KeyPress-key></code> 的简化形式。该格式只用于可打印字符 (空格和小于符号不包括在内)
<code><Prefix-key></code>	在按住 <code>Prefix</code> 的同时, 按下 <code>key</code> 。Prefix 可选择 Alt、Shift 和 Control。注意, 也可同时使用多个 Prefix, 比如 <code><Control-Alt-key></code>

图 10.13 键盘事件格式

```

1 # Fig. 10.14: fig10_14.py
2 # Binding keys to keyboard events.
3
4 from Tkinter import *
5
6 class KeyDemo( Frame ):
7     """Demonstrate keystroke events"""
8
9     def __init__( self ):
10         """Create two Labels and bind keystroke events"""
11
12         Frame.__init__( self )

```



```
13     self.pack( expand = YES, fill = BOTH )
14     self.master.title( "Demonstrating Keystroke Events" )
15     self.master.geometry( "350x100" )
16
17     self.message1 = StringVar()
18     self.line1 = Label( self, textvariable = self.message1 )
19     self.message1.set( "Type any key or shift" )
20     self.line1.pack()
21
22     self.message2 = StringVar()
23     self.line2 = Label( self, textvariable = self.message2 )
24     self.message2.set( "" )
25     self.line2.pack()
26
27     # binding any key
28     self.master.bind( "<KeyPress>", self.keyPressed )
29     self.master.bind( "<KeyRelease>", self.keyReleased )
30
31     # binding specific key
32     self.master.bind( "<KeyPress-Shift_L>", self.shiftPressed )
33     self.master.bind( "<KeyRelease-Shift_L>",
34         self.shiftReleased )
35
36     def keyPressed( self, event ):
37         """Display the name of the pressed key"""
38
39         self.message1.set( "Key pressed: " + event.char )
40         self.message2.set( "This key is not left shift" )
41
42     def keyReleased( self, event ):
43         """Display the name of the released key"""
44
45         self.message1.set( "Key released: " + event.char )
46         self.message2.set( "This key is not left shift" )
47
48     def shiftPressed( self, event ):
49         """Display a message that left shift was pressed"""
50
51         self.message1.set( "Shift pressed" )
52         self.message2.set( "This key is left shift" )
53
54     def shiftReleased( self, event ):
55         """Display a message that left shift was released"""
56
57         self.message1.set( "Shift released" )
58         self.message2.set( "This key is left shift" )
59
60     def main():
61         KeyDemo().mainloop()
62
63 if __name__ == "__main__":
64     main()
```

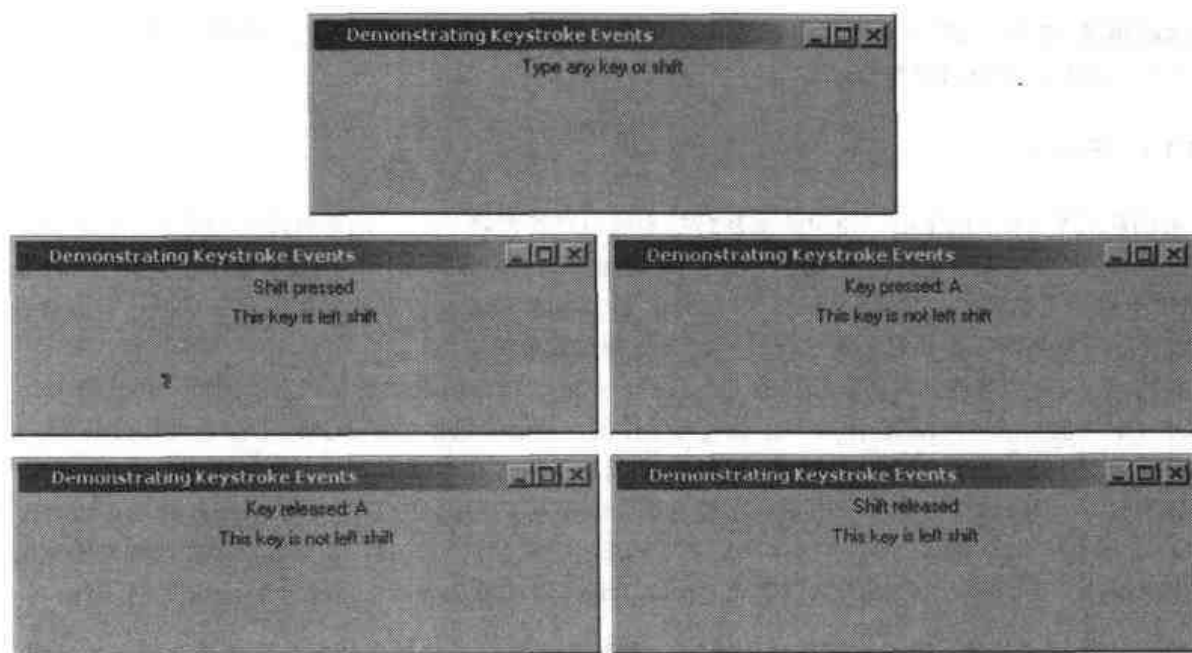


图 10.14 键盘事件演示

第 17~25 行创建两个标签，并对其进行 pack 操作。两个标签的名称分别是 line1 和 line2，用于显示与按键事件有关的信息。第 28~29 行将方法 keyPressed 和 keyReleased 分别绑定到<KeyPress>和<KeyRelease>事件。bind 方法（第 32~34 行）将左侧 Shift 键的<KeyPress-key>和<KeyRelease-key>事件分别关联到 shiftPressed 和 shiftReleased 方法。

用户按下或松开左侧 Shift 键，会分别执行 shiftPressed 方法（第 50~54 行）或 shiftReleased 方法（第 56~60 行），它们会在标签组件中显示恰当的消息。如果用户选择了不同于 Shift 的其他键，keyPressed 和 keyReleased 方法会在 line1 和 line2 中显示消息，指出生成事件的是哪一个键。注意，这两个方法使用 Event 对象的 char 属性获得按键的名称。

移植性提示 10.2 并不是所有系统都能区分左右 Shift 键。

10.10 布局管理器

布局管理器（Layout manager）用于排列 GUI 组件。大多数布局管理器都有可供程序员使用的基本布局能力，而不要求确切地知道每个 GUI 组件的具体位置及大小。用布局管理器来处理大多数设计细节，程序员可有效地节省时间和精力，将注意力集中于 GUI 的基本“外观与感觉”。图 10.15 总结了可用的布局管理器。

布局管理器	说明
Pack	按添加顺序放置组件
Grid	以行、列形式排列组件
Place	允许程序员指定组件和窗口的大小和位置

图 10.15 GUI 布局管理器

良好编程习惯 10.3 妥善选择布局管理器可使 GUI 编程更容易。编程之前，先画好设计图，再选择最合适的布局管理器。

常见编程错误 10.2 在同一个容器中使用多种类型的布局管理器，一旦 Tkinter 试图协调不同管理器的不同需求，就会导致应用程序死机。

10.10.1 Pack

前面所有的 GUI 例子中，使用的都是最基本的布局管理器 Pack。除非程序员指定了不同顺序，否则 Pack 会按它们在程序中列出的顺序，在一个“容器”中从上到下地放置 GUI 组件。所谓容器，是一个 GUI 组件，可在其中放置其他组件。容器对于 GUI 组件的布局管理非常有用。一旦抵达容器边缘，容器就会尽可能展开。如果容器无法展开，就无法看见其余的组件。

将组件 pack 到容器中时，程序员有几个选择。side 选项指明组件要靠在容器的哪一边。将 side 设为 TOP（默认值），组件会靠在顶部，并垂直向下排列。其他值包括 BOTTOM、LEFT 和 RIGHT 等。fill 选项可设为 NONE（默认）、X、Y 或 BOTH，它们指定了组件在容器中应该占据的空间。将 fill 设为 X、Y 或 BOTH，可确保组件在容器中占据分配给它的全部空间（在指定的方向上）。expand 选项可设为 YES 或 NO（1 或 0）。默认值是 NO。如 expand 设为 YES，组件会自动扩展，填充容器中任何额外的空间。padx 和 pady 选项可围绕组件插入空白填充。pack_forget 方法可从容器中删除一个 pack 好的组件。

良好编程习惯 10.4 使用布局管理器前，仔细阅读 Python 联机文档提供的布局管理器选项和方法列表。

常见编程错误 10.3 pack 方法按定义顺序将各组件放到一个容器中；所以，如错误定义顺序，会导致不可预料的结果。相反，如果在放置组件时，采用了为 side，expand，fill，padx 和 pady 指派值的方式，就可以忽略其定义顺序，保证获得所需结果。

图 10.6 创建 4 个按钮，并用 Pack 布局管理器把它们添加到应用程序。这个例子对按钮的位置及大小进行了调整。

```

1 # Fig. 10.16: fig10_16.py
2 # Pack layout manager demonstration.
3
4 from Tkinter import *
5
6 class PackDemo( Frame ):
7     """Demonstrate some options of Pack"""
8
9     def __init__( self ):
10         """Create four Buttons with different pack options"""
11
12         Frame.__init__( self )
13         self.master.title( "Packing Demo" )
14         self.master.geometry( "400x150" )
15         self.pack( expand = YES, fill = BOTH )
16
17         self.button1 = Button( self, text = "Add Button",
18                               command = self.addButton )
19
20         # Button component placed against top of window
21         self.button1.pack( side = TOP )
22
23         self.button2 = Button( self,
24                               text = "expand = NO, fill = BOTH" )
25
26         # Button component placed against bottom of window
27         # fills all available vertical and horizontal space
28         self.button2.pack( side = BOTTOM, fill = BOTH )
29
30         self.button3 = Button( self,
31                               text = "expand = YES, fill = X" )
32
33         # Button component placed against left side of window
34         # fills all available horizontal space
35         self.button3.pack( side = LEFT, expand = YES, fill = X )

```

```

36
37     self.button4 = Button( self,
38         text = "expand = YES, fill = Y" )
39
40     # Button component placed against right side of window
41     # fills all available vertical space
42     self.button4.pack( side = RIGHT, expand = YES, fill = Y )
43
44     def addButton( self ):
45         """Create and pack a new Button"""
46
47         Button( self, text = "New Button" ).pack( pady = 5 )
48
49     def main():
50         PackDemo().mainloop()
51
52     if __name__ == "__main__":
53         main()

```

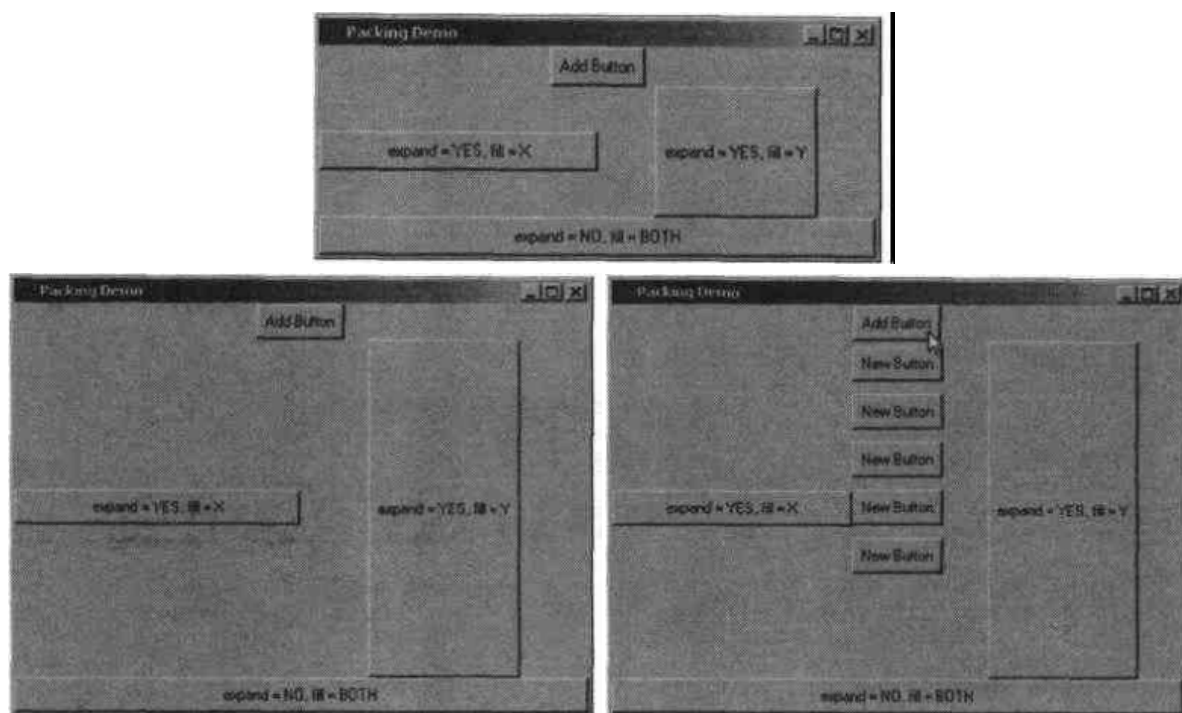


图 10.16 Pack 布局管理器演示

Frame 构造函数 (第 12 行) 允许基类执行在添加组件之前需要的任何初始化操作。title 方法 (第 13 行) 在 GUI 中显示标题。geometry 方法 (第 14 行) 将宽度和高度分别设为 300 和 150 像素。expand 和 fill 选项 (第 15 行) 分别设为 YES 和 BOTH, 确保 packDemo 这个 GUI 会充满整个窗口。第二个屏幕截图显示了用鼠标拖动 GUI 的边框以改变它的大小的结果。

第 17~42 行创建并 pack 了 4 个按钮, 分别为每个按钮指定不同的 pack 选项。Pack 布局管理器按照它们在程序中出现的顺序, 将每一项都放到顶级组件上。通过为 side, expand 和 fill 选项指定值, 可确保按钮按照屏幕截图的样子出现。pack 方法 (第 21 行) 依据 side 选项的指示, 将 button1 放在容器顶部。由于 fill 和 expand 默认都是 FALSE, 所以按钮组件将保持其默认大小。下一个组件 button2 (第 28 行) 放到容器底部。由于 fill 选项设为 BOTH, 所以按钮控件应该占据由容器分配给它的全部空间。对于 button3, expand 选项设为 YES (第 35 行), pack 方法会把这个组件放到容器左侧。expand 选项要求该按钮占据容器中任何可用的空间。将 fill 设为 X, 会导致该按钮填充由容器分配给它的所有水平空间。最后一个组件是 button4, 它放置于容器右侧。由于 fill 设为 Y, 所以按钮会填充由容器分配给它的所有垂直空间。

只有一个按钮 (button1) 指定了回调方法。一旦用户按下该按钮, addButton 方法 (第 44~47 行) 就会创建和 pack 一个新按钮。新建的按钮水平放置到 button1 下, 而且每次都会在垂直方向为新按钮添加 5 个像素的空白填充 (pady = 5)。

10.10.2 Grid

Grid 布局管理器将容器分割为一个网格, 使组件能以行、列形式放置。组件在网格中的位置由其 row 和 column 值决定; 网格中的每个单元格都可包含一个组件。行、列编号从 0 开始。如果没有指定 row 选项, 组件就放到第一个空行, 而且默认 column 值是 0。如果省略 column 选项, column 值将默认为 0。程序员可设置网格中的初始行列数, 具体做法是在 grid 构造函数中指定两个选项。此外, 还可分别调用 rowconfigure 和 columnconfigure 方法来设置行和列。图 10.17 在 GUI 中放入几种类型的组件, 对 Grid 布局管理器进行了演示。

```

1 # Fig. 10.17: fig10_17.py
2 # Grid layout manager demonstration.
3
4 from Tkinter import *
5
6 class GridDemo( Frame ):
7     """Demonstrate the Grid geometry manager"""
8
9     def __init__( self ):
10         """Create and grid several components into the frame"""
11
12         Frame.__init__( self )
13         self.master.title( "Grid Demo" )
14
15         # main frame fills entire container, expands if necessary
16         self.master.rowconfigure( 0, weight = 1 )
17         self.master.columnconfigure( 0, weight = 1 )
18         self.grid( sticky = W+E+N+S )
19
20         self.text1 = Text( self, width = 15, height = 5 )
21
22         # text component spans three rows and all available space
23         self.text1.grid( rowspan = 3, sticky = W+E+N+S )
24         self.text1.insert( INSERT, "Text1" )
25
26         # place button component in first row, second column
27         self.button1 = Button( self, text = "Button 1",
28                               width = 25 )
29         self.button1.grid( row = 0, column = 1, columnspan = 2,
30                           sticky = W+E+N+S )
31
32         # place button component in second row, second column
33         self.button2 = Button( self, text = "Button 2" )
34         self.button2.grid( row = 1, column = 1, sticky = W+E+N+S )
35
36         # configure button component to fill all it allocated space
37         self.button3 = Button( self, text = "Button 3" )
38         self.button3.grid( row = 1, column = 2, sticky = W+E+N+S )
39
40         # span two columns starting in second column of first row
41         self.button4 = Button( self, text = "Button 4" )
42         self.button4.grid( row = 2, column = 1, columnspan = 2,
43                           sticky = W+E+N+S )
44
45         # place text field in fourth row to span two columns
46         self.entry = Entry( self )
47         self.entry.grid( row = 3, columnspan = 2,
48                          sticky = W+E+N+S )
49         self.entry.insert( INSERT, "Entry" )
50
51         # fill all available space in fourth row, third column

```

```

52     self.text2 = Text( self, width = 2, height = 2 )
53     self.text2.grid( row = 3, column = 2, sticky = W+E+N+S )
54     self.text2.insert( INSERT, "Text2" )
55
56     # make second row/column expand
57     self.rowconfigure( 1, weight = 1 )
58     self.columnconfigure( 1, weight = 1 )
59
60 def main():
61     GridDemo().mainloop()
62
63 if __name__ == "__main__":
64     main()

```

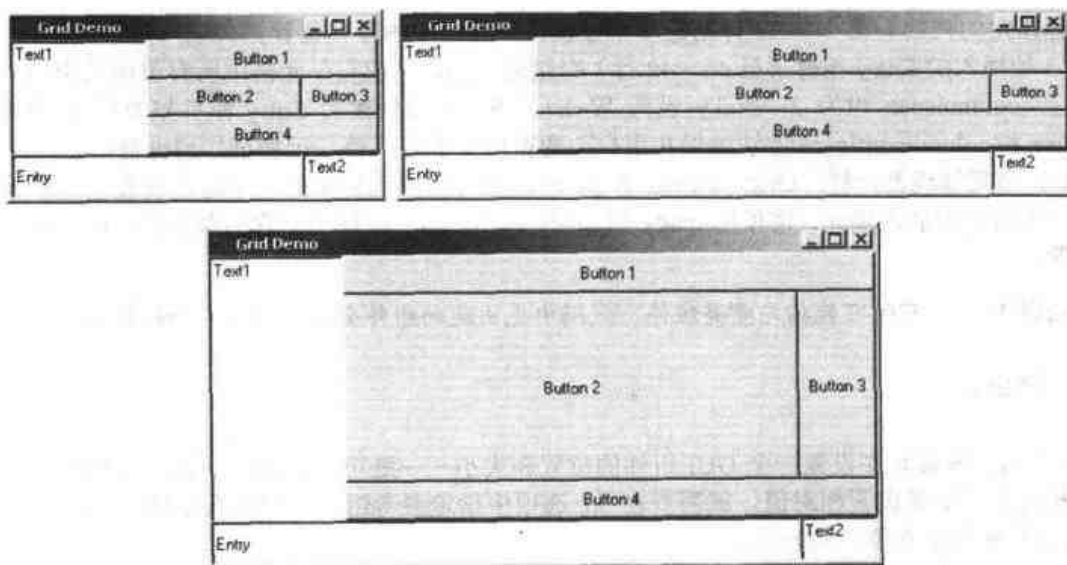


图 10.17 Grid 布局管理器演示

grid 方法 (第 18 行) 将顶级组件放到默认的行 0 和列 0 处。gridDemo 对象的 sticky 选项设为 W+E+N+S; 这会导致主帧扩展至填充完整个单元格。sticky 选项指定了组件的对齐方式, 以及组件是否展开以填充单元格。sticky 可选的值包括 W (西)、E (东)、N (北)、S (南)、NW (西北)、NE (东北)、SW (西南) 和 SE (东南) 的任意组合。例如, 假定将 sticky 设为 W+E, 就类似于在对组件进行 pack 处理时将 fill 设为 X——组件会从左一直向右伸展, 以填充单元格。如果将 sticky 设为 W+E+N+S, 就可获得与 Pack 布局管理器的 fill 值设为 BOTH 相同的效果。如果只为 sticky 指定一个值, 就类似于 Pack 的 side 选项 (组件会与指定的单元格边框对齐, 而不进行伸展)。第二个屏幕截图显示了用鼠标改变 GUI 大小后的效果。

Grid 管理器允许通过几个方法来控制组件在容器中的位置。其中, rowconfigure 和 columnconfigure 方法可分别更改行和列选项。例如, 在窗口大小改变时, 为了确保行 0 进行伸展, rowconfigure 方法 (第 16 行) 将 weight 选项设为 1。weight 选项指出一个行或列的相对缩放比例。针对一个行, 如果将 weight 设为 3, 那么它将以三倍于 weight 值为 1 的比例伸展。默认值是 0, 表明当用户改变窗口大小时, 单元格的大小保持不变。图 10.18 总结了最常用的 Grid 方法。

Grid 方法	说明
columnconfigure(column, options)	设置列选项, 如 minsize (最小长度)、pad (为列内最大的组件添加填充空距) 以及 weight
grid()	根据可选的关键字参数的说明, 将一个组件放到 Grid 中
grid_forget()	删除但不销毁一个组件
grid_remove()	删除一个组件, 存储与它对应的选项, 以便在重新插入组件时使用
grid_info()	将当前选项作为一个字典返回

Grid 方法	说明
<code>grid_location(x, y)</code>	以元组形式 (<i>column</i> , <i>row</i>) 返回最接近指定像素坐标的网格位置
<code>grid_size()</code>	以元组形式 (<i>column</i> , <i>row</i>) 返回网格的大小
<code>rowconfigure(row, options)</code>	设置行选项, 如 <i>minsize</i> (最小长度)、 <i>pad</i> (为行内最大的组件添加填充间距) 以及 <i>weight</i>

图 10.18 Grid 方法

第 20 行引入了 Text 组件, 它创建多行文本区域 `text1`。grid 方法 (第 23 行) 将 `text1` 组件插入网格, 并引入了关键字参数 `rowspan`。rowspan 选项设置一个组件在 GUI 中占据的行数。

`button1` 组件 (第 27 行) 占据两列, 这是由关键字参数 `columnspan` 指定的。columnspan 选项导致一个组件跨越指定的列数。第 27~43 行创建 4 个按钮, 并在指定的行列位置显式插入每个按钮。

在行 3 处插入的 Entry 组件 (第 46~49 行) 跨越两个列, 并填充单元格中所有可用的空间。在程序的第 47 行, `columnspan` 设为 2, `sticky` 设为 W+E+N+S——使创建的 Entry 组件填充行 3 的前两列。rowconfigure 和 columnconfigure 方法确保在用户改变窗口大小时, 第二行和第二列也自动展开。

和 Pack 布局管理器一样, Grid 的 `padx` 和 `pady` 选项可为单元格中的一个组件设置垂直和水平填充大小。为了在组件中插入填充, 请使用 `ipadx` 和 `ipady` 选项。如果组件小于它的单元格, 就默认在单元格内居中放置。

常见编程错误 10.4 有时可能指定重叠组件。代码中先出现的组件会被最近添加的组件遮住。

10.10.3 Place

Place 布局管理器允许设置一个 GUI 组件的位置和大小——既可是绝对值, 也可与另一个组件的相对位置和大小。如果设置相对值, 就需要在 `in_` 选项中指定参考组件, 它要么是插入组件的父 (默认设置), 要么和插入组件位于同一级。

Place 布局管理器要比其他管理器复杂。所以, 本节不打算详细讨论它, 只在图 10.19 列出了最常用的 Place 方法, 图 10.20 列出了常用的 `place` 和 `place_configure` 方法选项。欲知 Place 布局管理的详情, 请访问 www.python.org。

Place 方法	说明
<code>place()</code>	根据关键字参数的设置, 插入一个组件
<code>place_forget()</code>	删除但不销毁一个组件
<code>place_info()</code>	在一个字典中返回当前选项
<code>place_configure()</code>	根据关键字参数的设置, 定位一个组件

图 10.19 Place 方法

Place 选项	说明
<code>x</code>	指定组件的绝对水平位置
<code>y</code>	指定组件的绝对垂直位置
<code>relx</code>	相对于另一个组件, 指定组件的水平位置
<code>rely</code>	相对于另一个组件, 指定组件的垂直位置
<code>width</code>	指定组件的绝对宽度
<code>height</code>	指定组件的绝对高度
<code>relwidth</code>	相对于另一个组件, 指定组件的宽度
<code>relheight</code>	相对于另一个组件, 指定组件的高度
<code>in_</code>	指定一个参考组件。新插入的组件 (必须是参考组件的同级组件或者子) 将相对于它放置
<code>anchor</code>	指定组件的哪些部分“固定”于指定位置。可选值包括 NW (默认)、N、NE、E、SE、S、SW、W 和 CENTER

图 10.20 Place 选项

10.11 洗牌和发牌模拟

本节要通过随机数生成来开发一个洗牌和发牌模拟程序。以后可根据这个程序,实现具体的纸牌游戏。图 10.21 的程序开发了名为 Deck 的 GUI 类,它使用 Card 对象创建一副纸牌,其中包括 52 张牌。然后,允许用户单击"Deal card"(发牌)按钮发每一张牌。发出的每张牌都在一个标签中显示。用户任何时候都可单击"Shuffle cards"(洗牌)按钮进行洗牌。

Card 类(第 7~26 行)包含两个列表,即 faces 和 suits,分别存储所有可能的牌点和花色。类的构造函数(第 17~21 行)从 faces 列表接收一个字符串,并从 suits 列表接收另一个字符串。__str__ 方法(第 23~26 行)返回由纸牌的 face(牌点)构成的一个字符串,加上字符串" of",再加上纸牌的 suit(花色)。

```

1 # Fig. 10.21: fig10_21.py
2 # Card shuffling and dealing program
3
4 import random
5 from Tkinter import *
6
7 class Card:
8     """Class that represents one playing card"""
9
10    # class attributes faces and suits contain strings
11    # that correspond to card face and suit values
12    faces = [ "Ace", "Deuce", "Three", "Four", "Five",
13             "Six", "Seven", "Eight", "Nine", "Ten",
14             "Jack", "Queen", "King" ]
15    suits = [ "Hearts", "Diamonds", "Clubs", "Spades" ]
16
17    def __init__( self, face, suit ):
18        """Card constructor, takes face and suit as strings"""
19
20        self.face = face
21        self.suit = suit
22
23    def __str__( self ):
24        """String representation of a card"""
25
26        return "%s of %s" % ( self.face, self.suit )
27
28 class Deck( Frame ):
29     """Class to represent a GUI card deck shuffler"""
30
31    def __init__( self ):
32        """Deck constructor"""
33
34        Frame.__init__( self )
35        self.master.title( "Card Dealing Program" )
36
37        self.deck = [] # list of card objects
38        self.currentCard = 0 # index of current card
39
40        # create deck
41        for i in range( 52 ):
42            self.deck.append( Card( Card.faces[ i % 13 ],
43                                   Card.suits[ i / 13 ] ) )
44
45        # create buttons
46        self.dealButton = Button( self, text = "Deal Card",
47                                  width = 10, command = self.dealCard )
48        self.dealButton.grid( row = 0, column = 0 )
49
50        self.shuffleButton = Button( self, text = "Shuffle cards",
51                                      width = 10, command = self.shuffle )
52        self.shuffleButton.grid( row = 0, column = 1 )
53
54        # create labels
55        self.message1 = Label( self, height = 2,

```



```

56     text = "Welcome to Card Dealer!" )
57     self.message1.grid( row = 1, columnspan = 2 )
58
59     self.message2 = Label( self, height = 2,
60         text = "Deal card or shuffle deck" )
61     self.message2.grid( row = 2, columnspan = 2 )
62
63     self.shuffle()
64     self.grid()
65
66     def shuffle( self ):
67         """Shuffle the deck"""
68
69         self.currentCard = 0
70
71         for i in range( len( self.deck ) ):
72             j = random.randint( 0, 51 )
73
74             # swap the cards
75             self.deck[ i ], self.deck[ j ] = \
76                 self.deck[ j ], self.deck[ i ]
77
78         self.message1.config( text = "DECK IS SHUFFLED" )
79         self.message2.config( text = "" )
80         self.dealButton.config( state = NORMAL )
81
82     def dealCard( self ):
83         """Deal one card from the deck"""
84
85         # display the card, if it exists
86         if self.currentCard < len( self.deck ):
87             self.message1.config(
88                 text = self.deck[ self.currentCard ] )
89             self.message2.config(
90                 text = "Card #: %d" % self.currentCard )
91         else:
92             self.message1.config( text = "NO MORE CARDS TO DEAL" )
93             self.message2.config( text =
94                 "Shuffle cards to continue" )
95             self.dealButton.config( state = DISABLED )
96
97         self.currentCard += 1 # increment card for next turn
98
99     def main():
100         Deck().mainloop()
101
102     if __name__ == "__main__":
103         main()

```

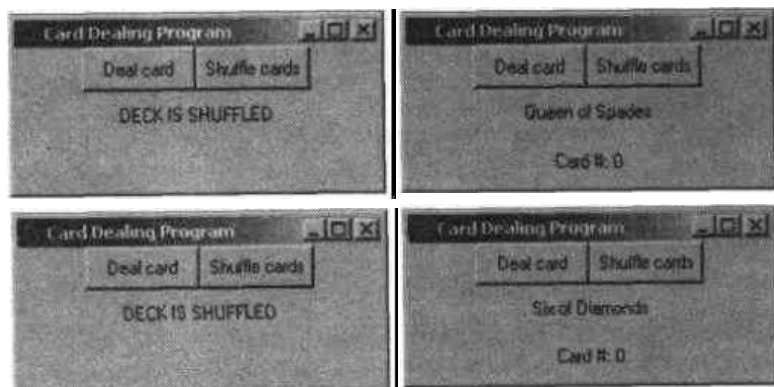


图 10.21 发牌程序

Deck 类（第 32~95 行）包括的元素有：一个由 52 个对象构成的 deck 列表；一个名为 currentCard 的整数，代表当前所发牌的索引；以及用于对这副牌进行操纵的一系列 GUI 组件。构造函数用 for 结构

在 `deck` 列表中填充 `Card` 对象 (第 41~43 行)。每个 `Card` 都进行实例化, 并用两个字符串初始化。这两个字符串中, 一个来自 `faces` 列表 (从 "Ace" 到 "King" 的字符串), 另一个来自 `suits` 列表 (包括 "Hearts", "Diamonds", "Clubs" 和 "Spades", 分别对应于红心、方块、梅花和黑桃)。注意, 这两个列表分别被引用为 `Card.faces` 和 `Card.suits`, 因为它们都是 `Card` 类的类属性。 $i \% 13$ 的计算结果肯定是从 0~12 的一个值 (即 `faces` 列表的 13 个下标), 而 $i / 13$ 的计算结果肯定是从 0~3 的一个值 (即 `suits` 列表的 4 个下标)。

如果用户单击 Deal card 按钮, `dealCard` 方法 (第 82~95 行) 会取得列表中的下一张牌。如果 `currentCard` 小于 52 (一副牌的总数), 第 87~88 行会在名为 `message1` 的标签中显示这张牌的 `face` 和 `suit`。`message2` 标签 (第 89~90 行) 显示牌的编号 (`currentCard`)。如果没有可发的牌 (即 `currentCard` 大于或等于 52), 会在 `message1` 中显示字符串 "NO MORE CARDS TO DEAL" (没有可以发的牌), 并在 `message2` 中显示字符串 "Shuffle cards to continue" (洗牌以继续)。

单击 Shuffle cards 按钮, 就会由 `shuffle` 方法 (第 66~80 行) 进行洗牌。该方法遍历全部 52 张牌 (0~51 的列表下标)。针对每张牌, 都会随机选择 0~51 的一个数字。接着, 当前的 `Card` 对象和随机选择的 `Card` 对象在列表中进行交换。这样, 遍历一次列表, 即可完成总共 52 次交换, 从而完成对 `Card` 对象列表的 "洗牌" 操作。洗牌结束后, 会在一个标签中显示 "DECK IS SHUFFLED" (洗牌完成)。

10.12 因特网和万维网资源

本节提供了有关在 Python 中使用 Tkinter 模块的几个网上资源。

faqs.com/knowledge_base/index.phtml/fid/264

该网页提供了有关 Tkinter 与 Python 的交互的问题与解答。

faqs.com/knowledge_base/index.phtml/fid/265

列出与事件处理有关的问题与解答。

www.pythonware.com/library/Tkinter/introduction

Fredrik Lundh 的 "An Introduction to Tkinter" 提供了与 Widget 类和事件处理有关的信息。

www.python.org/topics/Tkinter

通过这里提供的一些链接, 可获得 Tkinter、其他 Widget 类相关文档以及故障诊断提示。

www.csis.hku.hk/~kkto/doc-tkinter/tkinter/tkinter.html

Isaac K. K. To 的 "Building GUI Programs Using Tkinter: A Tkinter Manual" 提供了有关布局管理器、事件、Widget 类及其子类的信息。

第 11 章 图形用户界面组件（二）

学习目标

- 创建一个滚动项目列表，以便用户选择
- 创建滚动文本区域
- 创建菜单和弹出菜单
- 创建和操纵画布及滑杆

11.1 概述

本章继续学习 GUI。我们要讨论一些较高级的组件，并为以后构建复杂的 GUI 打下良好基础。

我们首先讨论“Python 巨元件”（Python Megawidgets, Pmw）。它是提供高级 GUI 组件的一个工具包，是基于由 Tkinter 模块提供的较小的组件而开发的。例如，一个 Pmw ScrolledListBox 组件允许用户从下拉列表中选择一项。接着讨论 ScrolledText 组件，它允许用户处理多行文本。最后要讨论菜单，Pmw 的 MenuBar 类可创建一个组件，帮助用户对菜单进行组织。

此外，本章还要介绍更多的 Tkinter 类。我们使用 Tkinter 的 Menu 类创建弹出式菜单（一种上下文相关菜单，右击含有弹出菜单的组件时即出现）。最后要讨论 Tkinter 的 Canvas 组件，它用于显示和处理文本、图像、线条和形状。Python 程序员还可选择其他许多 GUI 组件与工具包，所以在本章结束时，还要介绍其他几种工具包。

11.2 Pmw 简介

Pmw 包括一系列非常有用的 GUI 组件，它们基于 Tkinter 模块而构建。每个 Pmw 组件都合并了一个或多个 Tkinter 组件，目的是生成更有用的、更复杂的组件。事实上，每个 Tkinter 组件都可称为 Pmw 组件的一个“子组件”。例如，Pmw 的 ComboBox（组合框）组件合并了 Entry 和 Listbox 组件，从而创建一个更复杂的组件，既允许用户从列表框中选择一个项目，也允许他们在输入框中编辑文本。

Pmw 组件的每个子组件都可单独配置（可修改子组件的外观及功能）。Pmw 选项名称采取“*subcomponent_option*”的形式，程序员通过为这些选项指定值来配置一个 Pmw 组件。为了配置组件，可在组件创建时，在构造函数调用中传递选项值；也可在组件创建好之后，在 `configure` 方法调用中传递选项值。例如，以下语句可创建一个 ScrolledListBox 组件，并配置一个高度为 3 的 Listbox 子组件：

```
Pmw.ScrolledListBox( self, listbox_height = 3 )
```

以下语句可在现有的 Pmw TextDialog 组件中配置 text 组件的高度：

```
textdialog.configure( text_height = 10 )
```

虽然 Pmw 通过提供附加的组件，对 Tkinter 模块的功能进行了扩展，但 Pmw 并没有和 Python 捆绑在一起。要想下载这个产品，请访问 pmw.sourceforge.net。详细的安装步骤，请访问 Deitel & Associates 公司的网站（www.deitel.com）。

11.3 ScrolledListBox 组件

列表框（有时称为“下拉列表”）可提供一项项目列表，以便用户从中选择一项。Tkinter 的 Listbox

类 (它是 Widget 的一个派生类) 实现了列表框。

某些情况下, 由于列表中的项目个数太多, 以至于列表无法在屏幕上完整显示。为解决这个问题, 应允许用户在列表中滚动。为了实现滚动功能, 传统的做法是结合使用一个 Scrollbar (滚动条) 和一个 Listbox (列表框)。但 Pmw 为此提供了一个简单的解决方案, 即名为 ScrolledListBox (滚动列表框) 的一个 “巨元件”。

图 11.1 使用 ScrolledListBox 组件提供包含 4 个图像文件名的一个列表。用户选择一个图像文件名后, 程序会在 Label 中显示相应的图像。图 11.1 中的屏幕截图显示了做出选择之后的 ScrolledListBox 列表。

```

1 # Fig. 11.1: fig11_01.py
2 # ScrolledListBox used to select image.
3
4 from Tkinter import *
5 import Pmw
6
7 class ImageSelection( Frame ):
8     """List of available images and an area to display them"""
9
10    def __init__( self, images ):
11        """Create list of PhotoImages and Label to display them"""
12
13        Frame.__init__( self )
14        Pmw.initialise()
15        self.pack( expand = YES, fill = BOTH )
16        self.master.title( "Select an image" )
17
18        self.photos = []
19
20        # add PhotoImage objects to list photos
21        for item in images:
22            self.photos.append( PhotoImage( file = item ) )
23
24        # create scrolled list box with vertical scrollbar
25        self.listBox = Pmw.ScrolledListBox( self, items = images,
26            listBox_height = 3, vscrollmode = "static",
27            selectioncommand = self.switchImage )
28        self.listBox.pack( side = LEFT, expand = YES, fill = BOTH,
29            padx = 5, pady = 5 )
30
31        self.display = Label( self, image = self.photos[ 0 ] )
32        self.display.pack( padx = 5, pady = 5 )
33
34    def switchImage( self ):
35        """Change image in Label to current selection"""
36
37        # get tuple containing index of selected list item
38        chosenPicture = self.listBox.curselection()
39
40        # configure label to display selected image
41        if chosenPicture:
42            choice = int( chosenPicture[ 0 ] )
43            self.display.config( image = self.photos[ choice ] )
44
45    def main():
46        images = [ "bug1.gif", "bug2.gif",
47            "travelbug.gif", "buganim.gif" ]
48        ImageSelection( images ).mainloop()
49
50    if __name__ == "__main__":
51        main()

```

第一个 `ScrolledText` 中选择的文本。不必为 `ScrolledText` 组件绑定一种事件类型；相反，可借助一个外部事件（由不同 GUI 组件生成的事件）指出应在何时处理 `ScrolledText` 组件中的文本。例如，许多图形化电子邮件程序都设计了一个“发送”按钮，单击即可将邮件正文发送给收件人。该程序借鉴了这种做法，用一个按钮来生成外部事件，从而确定程序应在何时将选中的文本从左边的 `ScrolledText` 组件拷贝到右边的 `ScrolledText` 组件。

第 19~23 行创建并 `pack` 第一个名为 `text1` 的 `ScrolledText` 组件，其中包含一个 25 列、12 行的 `Text` 子组件。`ScrolledText` 组件的 `text_wrap` 选项决定了过长的文本行是否自动换行。如果被设为 `NONE`（默认值），就不显示超过限制的文本，只显示组件宽度范围内的文本。设为 `CHAR`，会以字符为最小单位进行断行。设为 `WORD`，则以单词为最小单位进行断行（即只在制表符和空格等空白字符处断行）。最后一个设置（即 `WORD`）实现了整词自动换行，这是许多文本编辑器的常见特性。

```

1 # Fig. 11.2: fig11_02.py
2 # Copying selected text from one text area to another.
3
4 from Tkinter import *
5 import Pmw
6
7 class CopyTextWindow( Frame ):
8     """Demonstrate ScrolledTexts"""
9
10    def __init__( self ):
11        """Create two ScrolledTexts and a Button"""
12
13        Frame.__init__( self )
14        Pmw.initialise()
15        self.pack( expand = YES, fill = BOTH )
16        self.master.title( "ScrolledText Demo" )
17
18        # create scrolled text box with word wrap enabled
19        self.text1 = Pmw.ScrolledText( self,
20            text_width = 25, text_height = 12, text_wrap = WORD,
21            hscrollmode = "static", vscrollmode = "static" )
22        self.text1.pack( side = LEFT, expand = YES, fill = BOTH,
23            padx = 5, pady = 5 )
24
25        self.copyButton = Button( self, text = "Copy >>>",
26            command = self.copyText )
27        self.copyButton.pack( side = LEFT, padx = 5, pady = 5 )
28
29        # create uneditable scrolled text box
30        self.text2 = Pmw.ScrolledText( self, text_state = DISABLED,
31            text_width = 25, text_height = 12, text_wrap = WORD,
32            hscrollmode = "static", vscrollmode = "static" )
33        self.text2.pack( side = LEFT, expand = YES, fill = BOTH,
34            padx = 5, pady = 5 )
35
36    def copyText( self ):
37        """Set the text in the second ScrolledText"""
38
39        self.text2.settext( self.text1.get( SEL_FIRST, SEL_LAST ) )
40
41    def main():
42        CopyTextWindow().mainloop()
43
44    if __name__ == "__main__":
45        main()

```

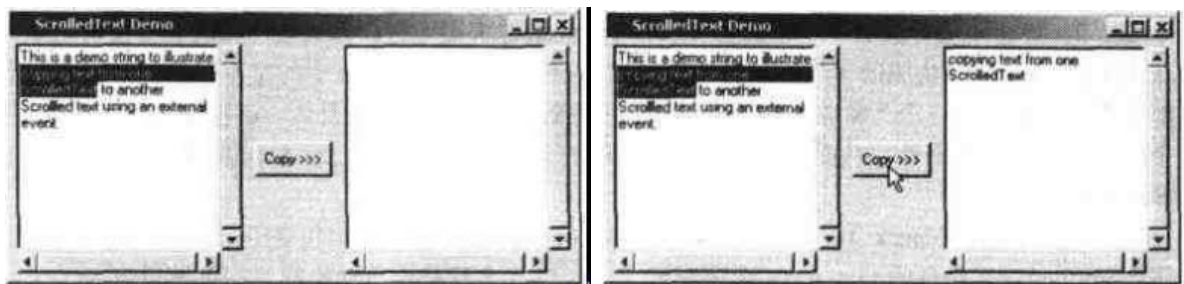


图 11.2 将文本从一个组件拷贝到另一个

界面知识 11.2 要使 Text 组件具有自动换行功能，请将 `text_wrap` 选项设为 `WORD`，而不是 `NONE`。

第 25~27 行创建并 `pack` 一个按钮，名为 `copyButton`，并为其绑定回调方法 `copyText`。第 30~34 行创建并 `pack` 第二个 `ScrolledText` 组件，即 `text2`。第 30 行将 `text2` 的 `text_state` 选项设为 `DISABLED`，它禁止 `insert` 和 `delete` 调用，使这个文本区域不可编辑。

若用户单击 `copyButton` 按钮，会执行 `copyText` 方法（第 36~39 行）。该方法调用组件的 `get` 方法，从 `text1` 获得用户输入的文本。`get` 方法取得两个参数，它们指定了要从组件中获得的文本范围。第 39 行通过指定一个范围，获得在 `text1` 中选定的文本。这个范围开始于选定文本的起始处（`SEL_FIRST`），终止于选定文本的结束处（`SEL_LAST`）。`setText` 方法删除组件中当前的文本，再插入作为一个参数由方法接收的文本。在这个例子中，`setText` 将 `get` 方法返回的文本插入 `text2`。如果用户尚未选定任何文本，程序会产生一个 `TclError` 异常，并在一个错误对话框中显示错误。第 7 章简要讨论了异常。在第 12 章中，将详细讨论如何处理异常（即防止程序自动显示错误对话框）。

11.5 MenuBar 组件

菜单是 GUI 不可分割的一部分，因为它们包含了一个选项列表。一旦选择某个选项，应用程序就会采取特定的行动。通过隐藏大量组件（按钮、链接等），菜单简化了 GUI 的外观。简单的 Tkinter GUI 要用 `Menu` 组件创建菜单。`Pmw` 模块则提供了更高级的 `MenuBar` 类，其中包含用于管理“菜单栏”（菜单的一种容器）所需的方法。

界面知识 11.3 菜单通过减少用户一次看到的组件数量，使 GUI 显得非常简洁。

菜单项是一个包含在菜单内部的 GUI 组件。用户一旦选择它，就会执行一项特定的动作。菜单项具有多种形式。其中，`command` 菜单项可发起一项动作。选择一个 `command` 菜单项，应用程序会调用该方法的回调方法。`checkboxbutton` 菜单项可在开和关两种状态之间切换。选择一个 `checkboxbutton` 菜单项，会在它的左侧显示一个勾号。用户可同时选择多个 `checkboxbutton`（也就是，它们不互相排斥）。如果选择一个已经勾选的 `checkboxbutton`，就相当于撤选操作，勾号会被清除。

`radiobutton` 菜单项也能在开和关状态之间切换。将多个 `radiobutton` 菜单项分为一组，用户就每次只能从中选择一个。选择一个 `radiobutton` 菜单项后，会在菜单项左侧显示一个勾号。如果从同一组中选择另一个菜单项，与原来所选菜单项对应的勾号会被清除。类似于 `radiobutton`（参见第 10 章），`radiobutton` 菜单项也要由一个共享变量进行逻辑性分组。

`separator` 菜单项会在菜单中显示一条横线。`cascade` 菜单项则会显示一个子菜单（或级联菜单），它可提供更多的菜单项，以使用户选择。

界面知识 11.4 `separator` 菜单项可用于从视觉上对相关的菜单项进行分组。

菜单栏包含了菜单项和子菜单。一旦单击某个菜单，菜单就会展开，并显示出菜单项和子菜单列表。单击一个菜单项，会生成一个事件。图 11.3 的程序创建了菜单和菜单项，以使用户更改一行文本的属性。

这个程序还演示了如何实现“气球”(或称“工具提示”),它能显示对菜单和菜单项的描述文字。将鼠标指针移到含有“气球”的菜单或菜单项上方,程序会自动弹出一条指定的帮助消息。

```

1  # Fig. 11.3: fig11_03.py
2  # MenuBars with Balloons demonstration.
3
4  from Tkinter import *
5  import Pmw
6  import sys
7
8  class MenuBarDemo( Frame ):
9      """Create window with a MenuBar"""
10
11     def __init__( self ):
12         """Create a MenuBar with items and a Canvas with text"""
13
14         Frame.__init__( self )
15         Pmw.initialise()
16         self.pack( expand = YES, fill = BOTH )
17         self.master.title( "MenuBar Demo" )
18         self.master.geometry( "500x200" )
19
20         self.myBalloon = Pmw.Balloon( self )
21         self.choices = Pmw.MenuBar( self,
22             balloon = self.myBalloon )
23         self.choices.pack( fill = X )
24
25         # create File menu and items
26         self.choices.addmenu( "File", "Exit" )
27         self.choices.addmenuitem( "File", "command",
28             command = self.closeDemo, label = "Exit" )
29
30         # create Format menu and items
31         self.choices.addmenu( "Format", "Change font/color" )
32         self.choices.addcascademenu( "Format", "Color" )
33         self.choices.addmenuitem( "Format", "separator" )
34         self.choices.addcascademenu( "Format", "Font" )
35
36         # add items to Format/Color menu
37         colors = [ "Black", "Blue", "Red", "Green" ]
38         self.selectedColor = StringVar()
39         self.selectedColor.set( colors[ 0 ] )
40
41         for item in colors:
42             self.choices.addmenuitem( "Color", "radiobutton",
43                 label = item, command = self.changeColor,
44                 variable = self.selectedColor )
45
46         # add items to Format/Font menu
47         fonts = [ "Times", "Courier", "Helvetica" ]
48         self.selectedFont = StringVar()
49         self.selectedFont.set( fonts [ 0 ] )
50
51         for item in fonts:
52             self.choices.addmenuitem( "Font", "radiobutton",
53                 label = item, command = self.changeFont,
54                 variable = self.selectedFont )
55
56         # add a horizontal separator in Font menu
57         self.choices.addmenuitem( "Font", "separator" )
58
59         # associate checkbox menu item with BooleanVar object
60         self.boldOn = BooleanVar()
61         self.choices.addmenuitem( "Font", "checkboxbutton",
62             label = "Bold", command = self.changeFont,
63             variable = self.boldOn )
64
65         # associate checkbox menu item with BooleanVar object
66         self.italicOn = BooleanVar()
67         self.choices.addmenuitem( "Font", "checkboxbutton",

```

```

68         label = "Italic", command = self.changeFont,
69         variable = self.italicOn )
70
71     # create Canvas with text
72     self.display = Canvas( self, bg = "white" )
73     self.display.pack( expand = YES, fill = BOTH )
74
75     self.sampleText = self.display.create_text( 250, 100,
76         text = "Sample Text", font = "Times 48" )
77
78     def changeColor( self ):
79         """Change the color of the text on the Canvas"""
80
81         self.display.itemconfig( self.sampleText,
82             fill = self.selectedColor.get() )
83
84     def changeFont( self ):
85         """Change the font of the text on the Canvas"""
86
87         # get selected font and attach size
88         newFont = self.selectedFont.get() + " 48"
89
90         # determine which checkbutton menu items selected
91         if self.boldOn.get():
92             newFont += " bold"
93
94         if self.italicOn.get():
95             newFont += " italic"
96
97         # configure sample text to be displayed in selected style
98         self.display.itemconfig( self.sampleText, font = newFont )
99
100     def closeDemo( self ):
101         """Exit the program"""
102
103         sys.exit()
104
105     def main():
106         MenuBarDemo().mainloop()
107
108 if __name__ == "__main__":
109     main()

```

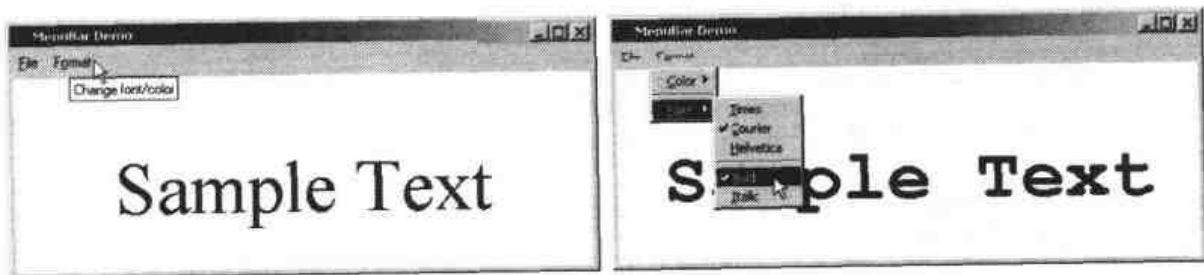


图 11.3 含有 Balloon 的 MenuBar

第 20 行创建了 myBalloon，这是一个 Pmw Balloon 组件。第 21~23 行创建并 pack 一个 MenuBar 组件，名为 choices。balloon 选项指定将一个 Balloon 组件与菜单项连接到一起。第 26~34 行则构建程序的菜单栏。addmenu 方法（第 26 行）为 choices 添加一个新菜单。方法的第一个参数（"File"）是菜单名称。第二个参数（"Exit"）包含了要在菜单的工具提示（气球）中显示的文本。一旦用户将鼠标指针停留在 File 菜单上方，程序就会在指针旁边的一个浮动标签中显示该文本。

第 27~28 行调用 addmenuitem 方法，在 File 菜单中插入一个 command 菜单项。该方法需要两个参数：菜单项所属的菜单的名称，以及菜单项的类型。这个例子在 File 菜单中添加了 Exit 菜单项。方法的关键字参数 label 指定了菜单项的文本。关键字参数 command 则指定了菜单项的回调方法。一旦从 File 中选择 Exit，回调方法 closeDemo（第 100~103 行）就会退出程序。

第 31 行为 choices 菜单栏添加 Format 菜单。addcascademenu 方法 (第 34 行) 在现有的菜单中添加一个子菜单。该方法需要两个参数: 子菜单所属的那个菜单的名称以及子菜单的文本。如果打开 Format 菜单, 并从中选择 Color, 程序就会显示出 Color 子菜单。第 33~34 行为 Format 菜单添加一个 separator 菜单项和一个 Font 子菜单。separator 菜单项实际是一条横线, 用于分隔 Color 和 Font 这两个子菜单。

界面知识 11.5 菜单项按照它们添加的顺序出现。因此, 务必按正确顺序添加。

界面知识 11.6 菜单通常按添加的顺序从左到右排列。

第 37 行为示范文本定义一个颜色选项列表。第 38~39 行创建一个 StringVar 对象 (名为 selectedColor 的), 并将其初始化成颜色选项列表的第一个元素。第 41~44 行针对颜色列表中的每一项, 都为 Color 子菜单添加一个 radiobutton 菜单项。注意, 所有 radiobutton 菜单项都共享同一个回调方法 (changeColor) 以及同一个变量 (selectedColor)。一旦用户选择一个菜单项, selectedColor 的值就会变成该项的文本值, 并调用 changeColor 方法。selectedColor 变量是同一组内所有 radiobutton 菜单项共用的。

第 51~54 行针对字体列表中的每一项, 都在 Format 菜单的 Font 子菜单中添加一个相应的 radiobutton 菜单项。每个 radiobutton 菜单项共享同一个回调方法 (changeFont) 和同一个变量 (selectedFont)。

第 57 行为 "Font" 子菜单添加 separator 菜单项。然后, 第 60~69 行为 Font 子菜单添加 "Bold" 和 "Italic" 这两个 checkbutton 菜单项。第 60 行和第 66 行创建了两个 BooleanVar 变量, 用于表示这些菜单项是否勾选。这些值通过关键字参数 variable 传给 addmenuitem 方法。尽管两个 checkbutton 菜单项共用一个回调方法 (changeFont), 但各自使用的 BooleanVar 变量不同。菜单项的 BooleanVar 变量的用途与 Tkinter Checkbutton 组件中相同。一旦用户选择菜单项, BooleanVar 的值会变成 1。如果用户撤选菜单项, BooleanVar 的值会变成 0。

第 72~73 行创建并 pack 一个 Tkinter Canvas (画布) 组件, 名为 display。它具有一个白色背景, 程序可在上面显示文本、线条和形状。Canvas 会显示一个“画布项目”, 它实际是一个可在 Canvas 组件上描绘的对象 (比如字符串或形状)。每个 Canvas 都有一个和画布项目对应的方法。每个方法都会创建一个画布项目, 并把它添加到 Canvas。例如, create_text 方法 (第 75~76 行) 可创建一个画布文本项目。该方法在 display 上描绘文本 "Sample Text", 并采用关键字参数 font 所指定的字体 ("Times 48")。11.7 节将更详细地讨论 Canvas 组件。

如果用户选择一个 Color 菜单项, changeColor 方法 (第 72~82 行) 会配置 sampleText, 使用 selectedColor 的值对 sampleText 进行填充 (上色) 处理。itemconfig 方法对 Canvas 上的项目进行配置。通过指定 fill 选项, 第 77~78 行将 sampleText 的颜色设置成所选的颜色。

如果用户从 Font 子菜单选择一个 radiobutton 菜单项, changeFont 方法 (第 84~98 行) 会更改 sampleText 的字体。第 98 行从 selectedFont 获取所需的字体名称。第 91~95 行确定是否从 Font 子菜单选择了任何 checkbutton 菜单项。如果是, 程序就在字体名称中附加指定的字形。然后, 第 92 行用指定的字体更新文本。

11.6 弹出菜单

目前许多计算机应用程序都支持“上下文关联弹出式菜单”。使用 Tkinter 的 Menu 类, 可方便地创建这种菜单。这种菜单所提供的选项特定于当时生成的“弹出触发事件”。在大多数系统上, 用户需要单击鼠标右键 (右击), 从而生成“弹出触发事件”。但使用 Tkinter, 必须为所需的触发器绑定一个回调方法, 只有这样, 才能指定一个弹出触发事件。

图 11.4 创建了一个 Menu, 它允许选择三种颜色之一作为 Frame 的背景色。右击 Frame, 程序会显示一个弹出菜单, 其中包含一个颜色列表。如果选择代表颜色的一个 radiobutton 菜单项, 程序会更改 Frame 的背景色。

```
1 # Fig. 11.4: fig11_04.py
```

```

2 # Popup menu demonstration.
3
4 from Tkinter import *
5
6 class PopupMenuDemo( Frame ):
7     """Demonstrate popup menus"""
8
9     def __init__( self ):
10         """Create a Menu but do not add it to the Frame"""
11
12         Frame.__init__( self )
13         self.pack( expand = YES, fill = BOTH )
14         self.master.title( "Popup Menu Demo" )
15         self.master.geometry( "300x200" )
16
17         # create and pack frame with initial white background
18         self.frame1 = Frame( self, bg = "white" )
19         self.frame1.pack( expand = YES, fill = BOTH )
20
21         # create menu without packing it
22         self.menu = Menu( self.frame1, tearoff = 0 )
23
24         colors = [ "White", "Blue", "Yellow", "Red" ]
25         self.selectedColor = StringVar()
26         self.selectedColor.set( colors[ 0 ] )
27
28         for item in colors:
29             self.menu.add_radiobutton( label = item,
30                                     variable = self.selectedColor,
31                                     command = self.changeBackgroundColor )
32
33         # popup menu on right-mouse click
34         self.frame1.bind( "<Button-3>", self.popUpMenu )
35
36     def popUpMenu( self, event ):
37         """Add the Menu to the Frame"""
38
39         self.menu.post( event.x_root, event.y_root )
40
41     def changeBackgroundColor( self ):
42         """Change the Frame background color"""
43
44         self.frame1.config( bg = self.selectedColor.get() )
45
46 def main():
47     PopupMenuDemo().mainloop()
48
49 if __name__ == "__main__":
50     main()

```

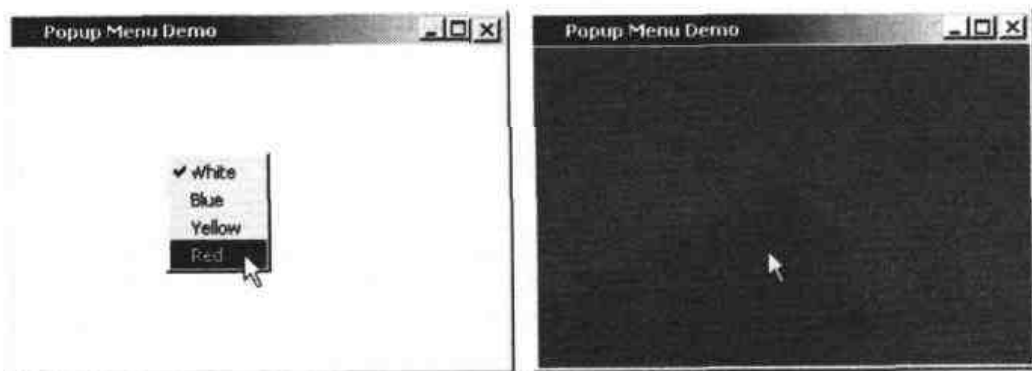


图 11.4 实现弹出菜单

Frame 构造函数的 `bg` 选项是指定了 Frame 背景色的一个字符串。第 18~19 行创建 `frame1`，并在其中填充一个白色背景。第 22 行创建一个 Tkinter 的 `Menu` 组件，名为 `menu`。注意，`Menu` 的 `tearoff` 选项

被设为 0。该设置会删除默认情况下作为 Menu 的第一项使用的虚线分隔符。第 28~31 行针对颜色列表中的每一项, 为 menu 添加相应的 radiobutton 菜单项。每个 radiobutton 菜单项都有相同的回调方法 (changeBackgroundColor) 以及相同的变量 (selectedColor)。

第 34 行将 popUpMenu 方法绑定到在 frame1 上发生的鼠标右击事件 (<Button-3>)。右击 frame1, 就会执行 popUpMenu 回调方法 (第 36~39 行)。注意, 第 39 行调用 Menu 的 post 方法, 从而在指定位置显示一个 Menu。该方法接收两个参数, 它们指定了顶级组件上的菜单显示位置。事件属性 x_root 和 y_root 对应于事件触发时鼠标指针的坐标。

一旦选择某个 radiobutton 菜单项, 就会执行 changeBackgroundColor 方法。该方法 (第 41~44 行) 调用 frame1 的 config 方法, 将新的 bg 指定为 selectedColor 的值 (第 44 行)。这个方法调用会改变 frame1 的背景颜色。

11.7 Canvas 组件

图 11.3 用一个 Canvas (画布) 显示格式化好的文本。Canvas 是一种 Tkinter 组件, 用于显示文本、图像、线条和形状。Canvas 继承于 Widget, 默认情况下是空白的。要在 Canvas 上显示一个项目, 程序必须创建“画布项目”。除非特别指定, 否则新项目会在现有项目的顶部描绘。

图 11.5 使用<B1-Motion>事件和一个 Canvas 来创建一个简单的绘图程序。在画布上拖动鼠标, 即可绘出一幅图。

```

1 # Fig. 11.5: fig11_05.py
2 # Canvas paint program.
3
4 from Tkinter import *
5
6 class PaintBox( Frame ):
7     """Demonstrate drawing on a Canvas"""
8
9     def __init__( self ):
10         """Create Canvas and bind paint method to mouse dragging"""
11
12         Frame.__init__( self )
13         self.pack( expand = YES, fill = BOTH )
14         self.master.title( "A simple paint program" )
15         self.master.geometry( "300x150" )
16
17         self.message = Label( self,
18                               text = "Drag the mouse to draw" )
19         self.message.pack( side = BOTTOM )
20
21         # create Canvas component
22         self.myCanvas = Canvas( self )
23         self.myCanvas.pack( expand = YES, fill = BOTH )
24
25         # bind mouse dragging event to Canvas
26         self.myCanvas.bind( "<B1-Motion>", self.paint )
27
28     def paint( self, event ):
29         """Create an oval of radius 4 around the mouse position"""
30
31         x1, y1 = ( event.x - 4 ), ( event.y - 4 )
32         x2, y2 = ( event.x + 4 ), ( event.y + 4 )
33         self.myCanvas.create_oval( x1, y1, x2, y2, fill = "black" )
34
35 def main():
36     PaintBox().mainloop()
37
38 if __name__ == "__main__":
39     main()

```



图 11.5 Canvas 绘图程序

第 17~19 行创建并 pack 一个标签,在其中显示操作指示文字。第 22~23 行创建并 pack 一个 Canvas 实例,名为 myCanvas。第 26 行将画布的鼠标拖动事件(<B1-Motion>)与方法 paint(第 28~33 行)绑定到一起。如果按住鼠标左键并移动鼠标,就会执行 paint 方法。该方法会在 myCanvas 这个画布上描绘一个椭圆。Canvas 的 create_oval 方法将创建名为 oval(椭圆)的一种“画布项目”,它的半径为 4,填充颜色为“black”,而且在当前鼠标指针位置居中(第 33 行)。

11.8 Scale 组件

Scale(滑杆)组件允许用户从一系列整数值中选择。Scale 类继承于 Widget。图 11.6 展示了一个水平滑杆,它允许通过一个滑块来选择数值。

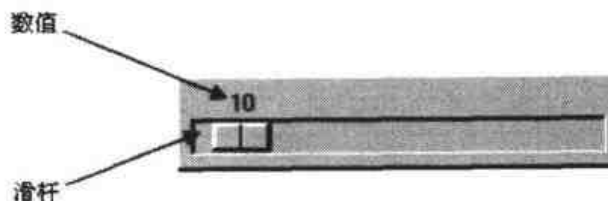


图 11.6 水平滑杆

Scale 既可选择水平方向,也可选择垂直方向。在水平 Scale 中,值从左到右逐渐增大。在垂直 Scale 中,值从上到下逐渐增大。

图 11.7 允许用户利用 Scale 组件指定要在 Canvas 上描绘的一个圆的大小。圆的直径由水平 Scale 控制。用户与 Scale 组件交互时,圆的大小会发生相应变化。

```
1 # Fig. 11.7: fig11_07.py
2 # Scale used to control the size of a circle.
3
4 from Tkinter import *
5
6 class ScaleDemo( Frame ):
7     """Demonstrate Canvas and Scale"""
8
9     def __init__( self ):
10         """Create Canvas with a circle controlled by a Scale"""
11
12         Frame.__init__( self )
13         self.pack( expand = YES, fill = BOTH )
14         self.master.title( "Scale Demo" )
15         self.master.geometry( "220x270" )
16
17         # create Scale
18         self.control = Scale( self, from_ = 0, to = 200,
19                             orient = HORIZONTAL, command = self.updateCircle )
20         self.control.pack( side = BOTTOM, fill = X )
21         self.control.set( 10 )
22
```

```

23     # create Canvas and draw circle
24     self.display = Canvas( self, bg = "white" )
25     self.display.pack( expand = YES, fill = BOTH )
26
27     def updateCircle( self, scaleValue ):
28         """Delete the circle, determine new size, draw again"""
29
30         end = int( scaleValue ) + 10
31         self.display.delete( "circle" )
32         self.display.create_oval( 10, 10, end, end,
33             fill = "red", tags = "circle" )
34
35     def main():
36         ScaleDemo().mainloop()
37
38     if __name__ == "__main__":
39         main()

```

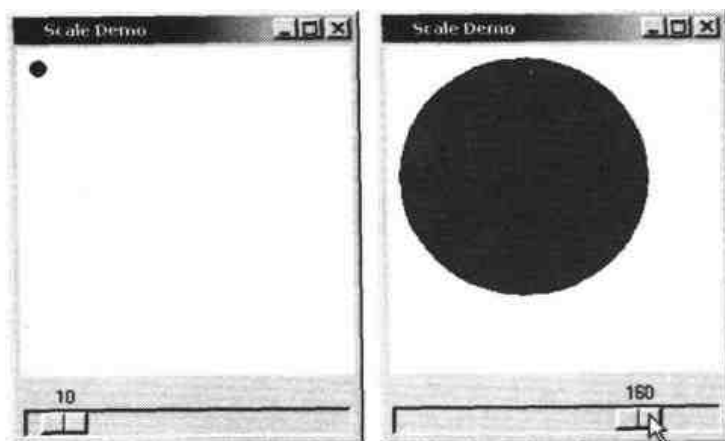


图 11.7 用 Scale 组件控制在 Canvas 上描绘的一个圆的大小

第 18~20 行创建并 pack 一个名为 control 的 Scale 组件，它用于改变圆的大小。构造函数的 orient 选项 (HORIZONTAL 或 VERTICAL) 决定了新的 Scale 实例的方向。from_ 和 to 选项指定了 Scale 组件的最小值和最大值。第 18~19 行的选项值创建了一个水平 Scale，而且最小值是 0，最大值是 200。Scale 的回调方法是 updateCircle。移动滑块以改变数值时，就会执行这个方法。注意，尽管 __init__ 没有在 display 上描绘任何东西，但程序一启动，圆就会在 display 上出现。这是因为在创建 Scale 时，会调用它的回调方法 (updateCircle)。第 21 行将 control 的值设为 10，所以程序启动时，直径为 10 的一个圆会出现在屏幕上。第 24~25 行创建并 pack 名为 display 的 Canvas 组件，并为它指定一个白色背景。

用户拖动滑块时，会执行 updateCircle 方法 (第 27~33 行)。回调方法的参数是 Scale 的当前值 (表示成一个字符串)。第 30 行把这个值转换成整数，为其加 10，再将结果保存到变量 end 中。

Canvas 的 delete 方法 (第 31 行) 先删除旧的圆，再描绘一个新的。delete 方法要取得一个参数——可能是一个 “item handle”，也可能是一个 “tag”。item handle 是用于对新描绘的项目进行标识的整数值，而 tag 是可在创建时连接到画布项目的一个名称。要想把 tag 和画布项目连接到一起，应向项目的 create 方法的 tags 选项传递一个字符串值。create_oval 方法 (第 32~33 行) 描绘一个椭圆，坐标是 (10, 10, end, end)，fill 选项值设为 “red”，tags 选项则设为 “circle”。坐标指定了这个椭圆的约束矩形的位置和大小。注意，Canvas 的 create_item 方法负责创建具体的画布项目，其中的 item 要替换成具体的项目名称，比如 arc, line, oval, rectangle, polygon, image, bitmap, text 和 window。

11.9 其他 GUI 工具包

其他还有许多用于 Python 的 GUI 工具包。PyGTK (www.daa.com.au/~james/pygtk) 为 Gimp ToolKit

(GTK) 组件集提供了面向对象的接口。GTK 的详情请参见 www.gtk.org。GTK 是一个高级的组件集，主要在 X Windows 系统中使用 (X Windows 是一种图形化系统，它为显示窗口化图形提供了一个公共接口)。PyGTK 是 GTK+ 的一部分，后者也是一个 Python 工具包，用于创建图形用户界面。

另一个流行的 GUI 工具包是 wxPython (www.wxpython.org)，它是一个 Python 扩展模块，提供了对 wxWindows 的访问途径 (wxWindows 是一个用 C++ 写成的 GUI 库)。这个工具包目前支持 Microsoft Windows 以及大多数 Unix 风格的操作系统。wxPython 这一 Python 模块封装了 wxWindows，提供清晰易用的接口，以使用户操纵其中的 wxWindows 类和方法。

PyOpenGL (pyopengl.sourceforge.net) 为 OpenGL (www.opengl.org) 库提供了一个 Python 接口。OpenGL 是开发交互式二维和三维图形应用程序的流行方案，支持 Microsoft Windows、MacOS 以及大多数 Unix 风格的系统。PyOpenGL 可与 Tkinter、wxPython 以及其他窗口图形库结合使用。第 24 章将详细讨论 PyOpenGL 模块。

第 12 章 异常处理

学习目标

- 理解异常以及错误处理
- 会用 try 语句对可能引发异常的代码进行定界
- 学会 raise 异常
- 会用 except 子句指定异常处理程序
- 会用 finally 子句释放资源
- 理解 Python 异常类层次结构
- 理解 Python 的跟踪机制
- 会创建程序员自定义的异常

12.1 概述

本章介绍“异常处理”(Exception Handling)。“异常”是程序执行期间发生的一个“特殊事件”。之所以称为“异常”，是因为尽管问题可能发生，但发生得并不频繁。这种特殊事件通常是一个错误（例如除以零，或者对两个不兼容的类型求和）；但有时，特殊事件可能是别的情况（例如 for 循环终止）。通过异常处理，应用程序能处理（或解决）异常。许多时候，进行了异常处理的程序可继续执行，好像没有发生过问题。较严重的问题则可能阻止程序继续正常执行。此时，程序可将问题通知给用户，再采取一种有控制的方式终止。本章介绍的内容可帮助程序员编写清晰的、健壮的以及容错性更佳程序。

Python 中异常处理的样式及细节基于 Modula-3 语言创始人的工作，它类似 C# 和 Java 中的机制。本章首先概述异常处理的概念，演示基本异常处理技术，再简单介绍了异常处理的类层次结构。

程序通常会在执行期间请求和释放“资源”（比如磁盘文件）。通常，这些资源的供应有限，或者一次只能供一个程序使用。针对这个问题，我们将演示异常处理机制中相应的那一部分功能，即让一个程序使用资源，然后保证该程序能释放资源，以供其他程序使用。

本章将举例说明 traceback 对象。Python 遇到异常时，就会创建这种对象。在本章最后，还要举例说明程序员应如何创建和使用自己的异常类。

12.2 引发异常

第 7 章介绍了 raise 语句，这一语句的作用是指出客户试图为对象属性分配一个无效的值。raise 语句表明出现了一个异常（比如函数无法成功完成），我们将其称为“引发”一个异常。

要想引发异常，最简单的形式就是输入关键字 raise，后跟要引发的异常的名称。异常名称标识出具体的类；Python 异常是那些类的对象。执行 raise 语句时，Python 会创建指定的异常类的一个对象。raise 语句还可指定对异常对象进行初始化的参数。为此，请在异常类的名称后添加一个逗号以及指定的参数（或者由参数构成的一个元组）。程序可根据异常对象的属性来获得与异常有关的更多信息。raise 语句具有多种形式，12.7 节将介绍 raise 的另一种不指定异常名称的形式。

测试和调试提示 12.1 用于初始化异常对象的参数可在异常处理程序中引用，以便执行恰当的任务。

测试和调试提示 12.2 即使不传递用于初始化异常对象的参数，也可引发一个异常。此时，只需知道平常发生这类异常的原因，处理程序就有充分把握来执行自己的任务。

到目前为止，本书只演示了如何用 raise 语句造成程序终止并打印一条错误消息（参见第 7 章）。

在后面的小节里，将演示如何检测一个发生的异常（称为“捕捉”异常），然后根据那个异常采取合适的行动（称为“处理”异常）。通过捕捉和处理异常，程序可知道何时发生了错误，并能采取相应的行动减小那个错误的影响。

12.3 异常处理

程序逻辑经常要检测各种条件，确定程序应如何继续。以下面的伪代码为例：

```
执行一个任务
如果上一个任务未能正确执行
    执行错误处理
执行下一个任务
如果上一个任务未能正确执行
    执行错误处理
...
```

上述伪代码首先执行一个任务，再检测该任务是否正确执行。如果不是，就执行错误处理。否则，伪代码将继续下一个任务。尽管这种形式的错误处理也许行得通，但将程序逻辑和错误处理逻辑混合到一起，会使程序难以阅读、修改、维护和调试——尤其是在大型应用程序中。事实上，如果存在许多发生频率并不高的潜在问题，将程序逻辑和错误处理混在一起会妨碍程序性能，因为程序必须不断测试附加条件，才能判断出是否能执行下一个任务。

通过异常处理，程序员可将异常处理代码从程序的执行流程中分离出来。这样，程序结构会变得更清楚，而且更易修改。程序员可自行决定对哪些异常进行处理。可选择处理所有类型的异常，特定类型的所有异常，或者一组相关类型的所有异常。这样的灵活性减少了错误被忽视的可能性，所以增强了程序的可靠性。

测试和调试提示 12.3 异常处理有助于增强程序的容错性。错误处理代码越容易编写，越有利于程序员。

有了不支持异常处理的编程语言，程序员通常将编写错误代码的工作推迟到后期完成，有时甚至会忘记编写错误处理代码。这会造成软件产品不够健壮。Python 能让程序员们一开始就轻松地进行异常处理。然而，程序员仍然必须将异常处理策略考虑到软件项目中。

软件工程知识 12.1 从设计阶段初期，就要将异常处理策略考虑到系统中。系统实现之后再添加有效的异常处理，往往很难。

软件工程知识 12.2 过去，程序员用多种技术来实现错误处理代码。异常处理则提供了一种统一的机制来处理错误。因此，许多程序员从事同一个大型项目时，相互之间可轻松理解对方的错误处理代码。

对于程序与可重用的软件元素进行交互时有可能发生的问题，也适合用异常处理机制进行处理。这些元素包括函数、类以及模块等等。这些软件元素不是在内部处理可能发生的问题，而是通过异常，向客户代码通知发生的问题。这样一来，程序员就可针对应用程序的具体情况，进行恰当的错误处理。

常见编程错误 12.1 退出程序可能导致资源（比如文件或网络连接）无法由其他程序使用。这称为“资源泄漏”。

性能提示 12.1 如果没有发生异常，异常处理代码对性能的影响极微（或者根本没有影响）。所以，相较于在程序逻辑中混合了错误处理逻辑的程序，实现了异常处理的程序效率更高。

软件工程知识 12.3 在复杂应用程序中，通常既包括预定义软件组件（比如在 Python 标准库中定义的

那些),也包括使用预定义组件的那个应用程序所特有的组件。如果预定义组件遇到问题,组件需要利用一种机制将问题通报给应用程序特有的组件——预定义组件事先不知道每个应用程序如何处理一个可能发生的问題。异常处理简化了软件组件的合成,并允许预定义组件将发生的问題报告给应用程序特有的组件(后者能采取一种应用程序特有的方式,对问題进行处理),使不同的组件能有效地配合运行。

软件工程知识 12.4 通常不要将异常显式地应用于常规控制流程(虽然也可以这样做)。否则会因为较难跟踪随后发生的大量异常,导致程序变得难以阅读和维护。

如果代码检测到一个不能处理的错误,便适合采用异常处理。这样的代码会“引发一个异常”。然而,并不保证肯定有一个异常处理程序——程序检测到异常时所执行的代码——对那种异常进行处理。如果有,异常会被“捕捉”(检测到),并得以“处理”。至于一个未被捕捉的异常,它的结果则取决于程序是一个 GUI 程序,还是一个控制台(非 GUI)程序;另外,还取决于程序是否运行于交互模式。在非 GUI 程序中,未被捕捉的异常会导致程序在打印一条错误消息后终止。如果 GUI 程序检测到一个未被捕捉的异常,程序会在控制台或对话框中显示错误消息(具体取决于所用的 GUI 包),然后继续执行。尽管 GUI 程序在未被捕捉的异常发生后能继续执行,但程序的行为可能和预期的不符。程序如果以交互模式运行,在侦测到一个未被捕捉的异常后,会显示错误消息,终止程序,再显示 Python 的交互式命令提示符。

Python 使用 try 语句实现异常处理。try 语句包围着可能引发异常的其他语句。try 语句以关键字 try 开头,后续一个冒号(:)和一个可能在其中引发异常的代码 suite(“suite”是 Python 用语,相当于“块”)。try 语句可指定一个或多个 except 子句,它们紧接在 try suite 之后。每个 except 子句都指定了零个或多个异常类名,它们代表要由 except 子句处理的异常类型。可在 except 子句(也称为“except 处理程序”)中指定一个标识符,程序用它引用被捕捉的异常对象。处理程序利用标识符从异常对象获取与异常有关的信息。如果 except 子句中没有指定异常类型,就称为“空白 except 子句”。这种子句会捕捉所有异常类型。在最后一个 except 子句之后,可选择性地添加一个 else 子句。如果 try suite 中的代码没有引发异常,就会执行 else 子句中的代码。如果 try 语句没有指定 except 子句,那就必须包含一个 finally 子句——该子句肯定会执行,无论是否发生异常。后文将讨论所有可能的子句组合。

常见编程错误 12.2 在 try 语句中同时包括 except 和 finally 子句是语法错误。可接受的组合形式只有: try/except, try/except/else 以及 try/finally。

一旦程序代码导致异常,或 Python 解释器检测到问題,代码或解释器就会“引发”一个异常。有的程序员将程序中发生异常的位置称为“引发点”——对于调试,这是一个重要的位置(详情参见 12.7 节)。异常是一些类的对象,这些类都是从 Exception 类继承的。^①如果在一个 try suite 中发生异常,这个 try suite 会立即“超时”(即立即终止),程序控制权会移交给 try suite 之后的第一个 except 处理程序(如果有的话)。接着,解释器搜索可对这种异常类型进行处理的第一个 except 处理程序。解释器为了确定相匹配的 except,需要将引发的异常的类型与每个 except 的异常类型比较,直到最终发现相匹配的为止。如类型完全一致,或引发的异常的类型是 except 处理程序的异常类型的一个派生类,就表明发生了匹配。如果 try suite 中没有发生异常,解释器将忽略用于 try 语句的异常处理程序,并执行 try 语句的 else 子句(如果有的话)。如果没有发生异常,或者某个 except 子句成功处理了异常,程序会从 try 语句之后的下一个语句恢复执行。如果在某个语句中发生了一个异常,但该语句不在一个 try suite 中,而是在一个函数中,那么包含那个语句的函数会立即终止,解释器会试图在(发出)调用(的)代码中查找一个封闭的 try 语句,这个过程称为“堆栈辗转开解”(Stack Unwinding),详情参见 12.7 节。

Python 使用“异常处理的终止模型”,因为引发异常的 try suite 会在异常发生后立即“过期”。^②

^① Python 异常也可能是字符串,以支持需要老版本 Python 解释器的程序。对于较新的 Python 版本(1.5.2 之后),应当首选基于类的异常处理技术。

^② 有的语言使用“异常处理的恢复模型”——完成异常处理后,控制权会返回异常引发点,并从那个位置恢复执行。

12.4 示例: DivideByZeroError

下面讨论一个简单的异常处理例子。图 12.1 的程序使用 `try`, `except` 和 `else` 来侦测和处理异常。程序提示用户输入两个数字, 分别代表一个除式中的分子和分母。输入这两个数字后, 程序针对用户输入的每个字符串调用 `float` 函数, 将其转换成浮点值。然后, 程序尝试将第一个值除以第二个值。如果输入的分母是 0, 当程序试图除以零时, 就会产生一个异常。另外, 如果用户输入任何一个值不是数字, 程序会显示一条消息, 要求用户输入数字。

```

1 # Fig. 12.1: fig12_01.py
2 # Simple exception handling example.
3
4 number1 = raw_input( "Enter numerator: " )
5 number2 = raw_input( "Enter denominator: " )
6
7 # attempt to convert and divide values
8 try:
9     number1 = float( number1 )
10    number2 = float( number2 )
11    result = number1 / number2
12
13 # float raises a ValueError exception
14 except ValueError:
15     print "You must enter two numbers"
16
17 # division by zero raises a ZeroDivisionError exception
18 except ZeroDivisionError:
19     print "Attempted to divide by zero"
20
21 # else clause's suite executes if try suite raises no exceptions
22 else:
23     print "%.3f / %.3f = %.3f" % ( number1, number2, result )

```

```

Enter numerator: 100
Enter denominator: 7
100.000 / 7.000 = 14.286

```

```

Enter numerator: 100
Enter denominator: hello
You must enter two numbers

```

```

Enter numerator: 100
Enter denominator: 0
Attempted to divide by zero

```

图 12.1 用 `try`, `except` 和 `else` 进行异常处理

讨论程序细节前, 先看看图 12.1 的示范输出窗口。第一个输出显示了一次成功的计算, 输入的分子是 100, 分母是 7, 在输出中显示了除法运算结果。在第二个输出中, 为第二个值输入的是字符串 "hello"。输入这个字符串并按下回车键, 程序就会显示一条消息, 指出必须输入数字。之所以会这样, 是因为 `float` 无法将字符串参数转换成浮点值, 所以函数引发了一个 `ValueError` 异常。程序捕捉到这个异常, 并显示恰当的提示消息。最后一个输出展示了除以零的结果。Python 解释器本身会检测除以零, 发现分母为零, 就引发 `ZeroDivisionError` 异常。程序捕捉到这个异常, 并显示一条消息, 指出除以零的问题。

接着要讨论为了获得示范输入/输出会话中的结果, 所需的用户交互及控制流程。用户要输入代表分子和分母的值。然后, 程序试图将用户输入的值转换成浮点值, 并用分母去除分子。第 8~11 行开始一个 `try` 语句, 它封装了可能引发异常的代码。注意, `try` suite 中的代码本身不包含任何 `raise` 语句, 所以表面上似乎不会“引发”异常。但事实上, `try` suite 中的语句通常调用的是其他有可能引发异常的代码。当然, 在某些时候, 也可由 `try` suite 中的语句自己引发异常 (比如在代码访问一个无效序列下标、字典键

或者对象属性时)。图 12.1 的 try suite 属于前一种情况,它调用了两次 float 函数(该函数可能引发 ValueError 异常),并执行了一次除法运算(它可能引发 ZeroDivisionError 异常)。

软件工程知识 12.5 要精心选择放入 try suite 中的代码小节。其中应有好几个语句都可能引发异常。尽量避免为可能引发异常的每个语句都单独使用一个 try 语句。但是,要想确保正确的异常处理,每个 try 语句都应尽可能封装一个足够小的代码区。这样一来,一旦发生异常,就可确切把握当时的背景,使 except 处理程序能正确地、有的放矢地处理异常。如果一个 try suite 中的多个语句都可能引发相同的异常类型,就必须用多个 try 语句确定每个异常的背景。

float 函数将用户输入的值转换成浮点值(第 9~10 行)。如果不能将字符串参数转换成浮点值,该函数就会引发 ValueError 异常。如果第 9~10 行正确转换了值(没有发生异常),第 11 行就会用分母去除分子,把结果指派给变量 result。如过分母为零,第 11 行将导致 Python 解释器引发一个 ZeroDivisionError 异常。如第 11 行没有引发异常,try suite 就结束执行。如果 try suite 中没有引发任何异常,第 14~15 行和第 18~19 行的 except 处理程序会被忽略,程序从 else suite(第 22~23 行)的第一个语句继续执行。这个 suite 实际上只有一行代码,作用就是打印除法运算结果。else suite 终止后,程序从整个 try 语句之后(即第 23 行之后)的第一个语句继续。在本例中,由于第 23 行之后没有更多的语句,所以整个程序终止。

常见编程错误 12.3 每个 try suite 与其第一个 except 处理程序之间、两个 except 处理程序之间、最后一个 except 处理程序和 else 子句之间或者 try suite 和 finally 子句之间,不允许出现任何语句,否则是语法错误。

测试和调试提示 12.4 尽管可在 try suite 中包含任意语句,但通常只包含可能引发异常的语句。在 else suite 中,则放置不会引发异常、而且只有在相应的 try suite 中没有发生异常的前提下才应执行的语句。

紧接在 try suite 之后的是两个 except 子句(也称为 except 处理程序或“异常处理程序”)。其中,第 14~15 行定义了用于 ValueError 的异常处理程序,第 18~19 行定义了用于 ZeroDivisionError 的异常处理程序。每个 except 子句都以关键字 except 开头,后续一个异常名称(指定了要由 except 子句处理的异常类型)和一个冒号(:)。异常处理代码位于 except 子句的主体中(即在缩进的一个代码 suite 中)。通常,一旦在 try suite 中发生异常,except 子句就会捕捉异常并对其进行处理。在图 12.1 中,第一个 except 子句指出它捕捉的是 ValueError 异常(由 float 函数引发)。第二个 except 子句指出它捕捉的是 ZeroDivisionError 异常(由解释器引发)。异常发生后,只有相匹配的 except 处理程序才会执行。本例的两个异常都会造成显示一条错误消息。程序控制抵达 except 处理程序的 suite 的最后一个语句时,解释器认为异常已得到处理,所以程序会从整个 try 语句之后的第一个语句(本例是程序的末尾)继续执行。

测试和调试提示 12.5 except 处理程序必须指定要捕捉的异常类名。只有针对默认的“全部捕捉”条件,才应使用一个空白的 except 处理程序。

在第二个输入/输出对话中,用户输入字符串“hello”作为分母。第 10 行执行时,float 无法将这个字符串值转换成浮点值,所以 float 引发一个 ValueError 异常,表明函数不能执行转换。异常发生后,try suite 会立即“超时”(终止)。接着,解释器尝试查找相匹配的 except 处理程序,首先考察的是第 14 行以 except 开头的那一个。解释器将所引发异常的类型(ValueError)与关键字 except 之后的关键字(也是 ValueError)进行比较。由于两者匹配,所以会执行这个异常处理程序。与此同时,解释器会忽略相应的 try suite 之后的其他所有异常处理程序。如果与第 14 行的 except 处理程序不匹配,解释器会顺序比较下一个 except 处理程序。并一直重复这个过程,直到发现相匹配的为止。

软件工程知识 12.6 在一个 except 子句中,可指定多个异常,只需在圆括号中使用由逗号分隔的一个异常名称序列即可。要用 except 子句指定多个异常,这些异常应以某种形式相互关联(例如都是因为算术计算错误而引发的异常)。将相关的异常分为一组,再为每一组都单独使用一个 except 子句。

图 12.1 的第 3 个输入/输出对话中,输入 0 作为分母。第 11 行执行时,解释器会引发 `ZeroDivisionError` 异常,指出检测到除以零试图。同样, `try suite` 遇到异常后立即终止,解释器会尝试查找匹配的 `except` 处理程序,首先从第 14 行开始。解释器将引发的异常的类型 (`ZeroDivisionError`) 与关键字 `except` 之后的关键字 (`ValueError`) 比较。在本例中,两者是不匹配的——因为 `ZeroDivisionError` 和 `ValueError` 不是相同的异常类型,而且 `ValueError` 不是 `ZeroDivisionError` 的基类。所以解释器转移到第 18 行继续比较,这便找到了一个匹配,并执行相应的异常处理程序。如果还有其他 `except` 处理程序,解释器会将其忽略。

12.5 Python 的 Exception 层次结构

本节概述了 Python 的几个异常类。所有异常都从基类 `Exception` 继承,而且都在 `exceptions` 模块中定义。Python 自动将所有异常名称放在内建命名空间中,所以程序不必导入 `exceptions` 模块即可使用异常。Python 定义了从 `Exception` 继承的 4 个主要的类,包括 `SystemExit`, `StopIteration`, `Warning` 以及 `StandardError`。其中,一旦引发而且没有捕捉 `SystemExit` 异常,程序执行就会终止。如果交互式会话遇到一个未被捕捉的 `SystemExit` 异常,会话就会终止。Python 使用 `StopIteration` 异常 (这是 Python 2.2 新增的) 来判断一个 `for` 循环何时抵达序列末尾。`Warning` 异常指出 Python 的一个特定元素将来有可能改变。例如,假定一个 Python 2.2 程序使用了名为 `yield` 的一个变量,那么 Python 会引发一个 `Warning` 异常,因为在 Python 未来的版本中,已将 `yield` 定义为关键字。`StandardError` 则是所有 Python “错误”异常 (比如 `ValueError` 和 `ZeroDivisionError`) 的一个基类。

图 12.2 总结了 Python 2.2 异常层次结构。对于任何版本的 Python,都可用以下语句显示该结构:

```
import exceptions
print exceptions.__doc__
```

Python 异常

```
Exception
  SystemExit
  StopIteration
  StandardError
    KeyboardInterrupt
    ImportError
    EnvironmentError
    IOError
    OSError
      WindowsError (注意: 只在 Windows 平台上定义)
  EOFError
  RuntimeError
    NotImplementedError
  NameError
    UnboundLocalError
  AttributeError
  SyntaxError
    IndentationError
      TabError
  TypeError
  AssertionError
  LookupError
    IndexError
    KeyError
  ArithmeticError
    OverflowError
    ZeroDivisionError
    FloatingPointError
  ValueError
    UnicodeError
  ReferenceError
  SystemError
  MemoryError
Warning
  UserWarning
```

Python 异常

```

DeprecationWarning
SyntaxWarning
OverflowWarning
RuntimeWarning

```

图 12.2 Python 异常层次结构

许多 `StandardError` 异常都可在程序运行时加以捕捉和处理，从而确保程序能继续运行。通过适当的编程，往往能避免这样的异常。例如，假定程序试图访问越界的序列下标，解释器会引发 `IndexError` 类型的异常。类似地，如果程序试图访问不存在的对象属性，会引发 `AttributeError` 异常。

异常类层次结构的一个优点在于，`except` 处理程序既可捕捉特定类型的异常，也可使用一个基类类型捕捉从它继承的所有类型的异常。例如，12.3 节讨论了空的 `except` 处理程序，它可捕捉所有类型的异常。如果在 `except` 处理程序中将异常类型规定为 `Exception`，同样可以捕捉所有异常（假定引发的异常是从 `Exception` 类继承的），这是因为 `Exception` 是所有异常类的基类。

为异常指派了继承层次结构之后，异常处理程序就可简单地捕捉一系列相关的异常。异常处理程序肯定能单独捕捉每个派生类异常，但更简单的做法是捕捉基类异常——只要以相同的方式来处理所有派生类异常。如果处理方式不同，那么还是需要单独捕捉每一个派生类异常。

常见编程错误 12.4 在特定 `try suite` 之后、最后一个 `except` 子句之前，设置一个空的 `except` 子句属于语法错误。

常见编程错误 12.5 在特定 `try suite` 之后，两个或更多的 `except` 子句指定完全相同的异常类型是逻辑错误。Python 会执行与引发的异常匹配的第一个 `except` 处理程序，忽略其他所有 `except` 处理程序。

常见编程错误 12.6 在其他 `except` 处理程序之前，用一个 `except` 处理程序捕捉类型为 `Exception` 的异常是逻辑错误，因为它会捕捉所有异常，使后面所有异常处理程序形同虚设。

要想确定 Python 以及标准/第三方组件何时引发异常，有时可能非常困难。例如，程序可能无法判断一个函数是否会引发特定的异常。不过，在语言参考以及标准库文档中^①，通常指明了会引发异常的前提条件。比如，图 12.1 演示了程序试图除以零时，Python 会引发一个 `ZeroDivisionError` 异常。相应地，语言参考的 5.6 节描述了除法运算符，并指出除以零会导致一个 `ZeroDivisionError` 异常。至于第三方组件，如果希望广为传播，并正式用于在软件开发中，理论上也应该包括这样的文档，并指明组件有可能引发哪些异常，以及为什么会引发这些异常。

软件工程知识 12.7 如果组件可能引发异常，组件文档就应对异常进行全面描述。另一方面，使用该组件的语句应放在 `try suite` 中，而且应在自己的程序中捕捉和处理那些异常。

12.6 finally 子句

程序经常动态地（也就是在执行时）请求和释放资源。例如，从硬盘上读取一个文件的程序会请求打开那个文件。如果请求成功，就能读取文件内容。操作系统通常会阻止多个程序同时操作同一个文件。因此，当程序结束处理文件时，程序通常会关闭文件（也就是释放资源），以便其他程序能使用这个文件。关闭文件有助于避免“资源泄漏”（即文件资源无法供其他程序使用，因为原来使用该文件的程序一直没把它关闭）。获得特定类型资源（如文件）的程序必须将资源明确归还给系统，以避免资源泄漏。

在 C 和 C++ 等语言中，程序员要自己负责动态内存管理，一种最常见的资源泄漏是“内存泄漏”。如果程序分配（获取）了内存，但在不需要之后没有解除分配（回收），就会发生这种情况。不过在 Python

^① 库参考文档网址是 www.python.org/doc/current/lib/lib.html；语言参考文档网址是 www.python.org/doc/current/ref/ref.html。

中,这通常不是一个问题,因为解释器会对正在执行的程序不需要的内存进行“垃圾回收”。但是,Python 中有可能发生其他类型的资源泄漏(比如前面提到的未关闭的文件)。

测试和调试提示 12.6 解释器不能完全消除内存泄漏。只要存在对一个对象的引用,解释器就不会对其进行垃圾回收。所以,如果程序员偶然保留了不需要的对象引用,仍会出现内存泄漏。

对于要求明确释放的大多数资源来说,往往存在一些与资源处理有关的异常。例如,一个文件处理程序可能在处理期间引发 IOError 异常。因此,文件处理代码通常应放在一个 try suite 中。无论程序是否成功处理文件,一旦不需要文件,都应将其关闭。

假定程序将所有资源请求和释放代码都放在一个 try suite 中。如果没有发生异常,try suite 会正常运行并释放资源。但是,如果发生异常,try suite 会在资源释放代码执行之前“过期”。虽然可在 except 处理程序中复制所有资源释放代码,但这样会使代码难以修改和维护。

Python 异常处理机制提供了 finally 子句,只要程序控制进入相应的 try suite,就必然会执行这个子句(无论 try suite 是成功执行,还是发生异常)。所以,对于在相应 try suite 中获取和处理的资源,finally suite 是放置资源回收代码的理想地点。如果 try suite 成功执行,finally suite 会在 try suite 终止后立即执行。如果在 try suite 中发生异常,那么在导致异常的那一行之后,会立即执行 finally suite。然后,由下一个封闭的 try 语句(如果有的话)来处理异常。

测试和调试提示 12.7 finally suite 包含的代码一般用于释放相应 try 块中获得的资源,这使 finally suite 成为消除资源泄漏的有效途径。

测试和调试提示 12.8 程序控制进入相应的 try 块时,finally 块没有执行的惟一原因是,应用程序在 finally 块能够执行之前就终止了。

性能提示 12.2 根据规则,资源一旦不需要,就应马上释放,以便资源能被立即重用,并允许其他程序访问那些资源。

软件工程知识 12.8 引发异常的代码要先释放代码中获取的任何资源,再引发异常。

图 12.3 证明,finally 子句肯定会执行,无论在相应的 try suite 中是否发生了异常。程序用两个函数来演示 finally,它们是 doNotRaiseException(第 4~14 行)和 raiseExceptionDoNotCatch(第 16~27 行)。主程序调用这些函数,以演示何时会执行 finally 子句。

```

1 # Fig. 12.3: fig12_03.py
2 # Using finally clauses.
3
4 def doNotRaiseException():
5
6     # try block does not raise any exceptions
7     try:
8         print "In doNotRaiseException"
9
10    # finally executes because corresponding try executed
11    finally:
12        print "Finally executed in doNotRaiseException"
13
14    print "End of doNotRaiseException"
15
16 def raiseExceptionDoNotCatch():
17
18    # raise exception, but do not catch it
19    try:
20        print "In raiseExceptionDoNotCatch"
21        raise Exception
22
23    # finally executes because corresponding try executed
24    finally:
25        print "Finally executed in raiseExceptionDoNotCatch"
```



```

26
27     print "Will never reach this point"
28
29 # main program
30
31 # Case 1: No exceptions occur in called function.
32 print "Calling doNotRaiseException"
33 doNotRaiseException()
34
35 # Case 2: Exception occurs, but is not handled in called function,
36 # because no except clauses exist in raiseExceptionDoNotCatch
37 print "\nCalling raiseExceptionDoNotCatch"
38
39 # call raiseExceptionDoNotCatch
40 try:
41     raiseExceptionDoNotCatch()
42
43 # catch exception from raiseExceptionDoNotCatch
44 except Exception:
45     print "Caught exception from raiseExceptionDoNotCatch " + \
46           "in main program."

```

```

Calling doNotRaiseException
In doNotRaiseException
Finally executed in doNotRaiseException
End of doNotRaiseException

Calling raiseExceptionDoNotCatch
In raiseExceptionDoNotCatch
Finally executed in raiseExceptionDoNotCatch
Caught exception from raiseExceptionDoNotCatch in main program.

```

图 12.3 总是执行 finally

主程序的第 33 行调用 `doNotRaiseException` 函数（第 4~14 行），该函数包含了一个 `try/finally` 组合。其中，`try suite`（第 8 行）输出一条消息。这个 `suite` 不会引发任何异常，所以程序控制正常抵达 `suite` 的末尾。接着执行的是 `finally` 子句的 `suite`（第 12 行），并输出一条消息。在这个时候，因为没有引发异常，所以程序控制会从 `finally suite` 之后的第一个语句继续。第 14 行的语句输出一条消息，指出已抵达函数末尾。接着，程序控制权会返回主程序。

常见编程错误 12.7 如果 `try` 语句既不包括 `finally` 子句，也不包括 `except` 子句，那就属于语法错误。如果 `try` 语句不包括任何 `except` 子句，那么必须包含一个 `finally` 子句。如果 `try` 语句不包括 `finally` 子句，那么至少要包括一个 `except` 子句。

主程序的第 40~41 行开始另一个 `try` 语句，它调用函数 `raiseExceptionDoNotCatch`（第 16~27 行）。`try` 语句允许主程序捕捉由 `raiseExceptionDoNotCatch` 引发的任何异常。在 `raiseExceptionDoNotCatch` 函数中，`try suite`（第 20~21 行）首先输出一条消息。接着，`try suite` 引发一个 `Exception`（第 21 行），造成这个 `try suite` 立即过期。由于这个 `try` 语句没有指定任何 `except` 子句，所以异常不会在 `raiseExceptionDoNotCatch` 函数中进行捕捉。正常的程序控制无法继续，除非异常得以捕捉和处理。因此，解释器会终止 `raiseExceptionDoNotCatch`，并让程序控制权返回至主程序。然而，在控制权回到主程序之前，`finally` 子句的 `suite`（第 25 行）会执行，并输出一条消息。之后，程序控制才回到主程序，`finally suite` 之后出现的任何语句都不会执行（比如第 27 行）。在主程序中，第 44~46 行的 `except` 处理程序捕捉异常，并显示一条消息，指出异常已在主程序中捕捉。

常见编程错误 12.8 在 `finally suite` 中引发异常是非常危险的。执行 `finally suite` 时，假如一个未被捕捉的异常正在等候处理，而 `finally suite` 又引发了一个新的、未被该 `suite` 捕捉的异常，那么第一个异常就会丢失，新异常则传递给下一个封闭的 `try` 语句。

测试和调试提示 12.9 在 `finally suite` 中，必须将可能引发异常的代码封闭到一个 `try` 语句中。这样可避

免丢失未被捕捉的，在 finally suite 执行之前发生的异常。

软件工程知识 12.9 如果 try 语句指定了一个 finally 子句，那么即使 try suite 由一个 return 语句终止，finally 子句的 suite 也会执行。执行完之后，才轮到通过 return 返回调用代码。

注意，在 finally 子句之后，程序具体从什么地方继续执行，取决于异常处理的状态。如果 try suite 成功完成，会执行 finally suite，再继续执行 finally suite 之后的下一个语句。如果 try suite 引发异常，执行了 finally suite 之后，继续执行的将是下一个封闭的 try 语句。封闭 try 既可能在调用函数中，也可能在它的某个调用者中。还有可能在一个 try suite 中嵌套另一个 try/except 组合。在这种情况下，外层 try 语句的异常处理程序会对内层 try 语句中未被捕捉的任何异常进行处理。

12.7 Exception 对象和跟踪

正如 12.5 节所示，从 Exception 类派生的异常数据类型可用零个或多个参数创建。这些参数常常用于为一个引发的异常格式化错误提示消息。只要 Python 为了响应一个 raise 语句而创建一个异常对象，就会将来自 raise 语句的任何参数放到异常对象 args 属性中。

发生异常时，Python 能“记住”引发的异常以及程序的当前状态。Python 还维护着 traceback（跟踪）对象，其中含有异常发生时与函数调用堆栈有关的信息。记住，异常可能在一系列嵌套较深的函数调用中引发。程序调用每个函数时，Python 会在“函数调用堆栈”的起始处插入函数名。一旦异常被引发，Python 会搜索一个相应的异常处理程序。如果当前函数中没有异常处理程序，当前函数会终止执行，Python 会搜索当前函数的调用函数，并以此类推，直到发现匹配的异常处理程序，或者 Python 抵达主程序为止。这一查找合适的异常处理程序的过程就称为“堆栈辗转开解”（Stack Unwinding）。解释器一方面维护着与放置于堆栈中的函数有关的信息，另一方面也维护着与已从堆栈中“辗转开解”的函数有关的信息。

测试和调试提示 12.10 traceback 展示了自异常发生之时起，一个完整的函数调用堆栈。这样一来，程序员就可查看导致异常的一系列函数调用。traceback 中的信息包括辗转开解的函数名称：在其中定义了函数的那个文件的名称以及程序遇到错误时的行号。traceback 中的最后一个行号是“引发点”（即初始异常的引发位置）。在它之前的一系列行号则是在 traceback 中每个函数的调用位置。

下一个例子（图 12.4）演示了异常对象的 args 属性以及异常对象的字符串表示。另外，这个例子还解释了如何访问 traceback 对象，以打印有关堆栈辗转开解的信息。讨论这个例子时，我们要跟踪调用堆栈上的函数，以便能澄清 traceback 对象以及堆栈辗转开解机制。

```

1 # Fig. 12.4: fig12_04.py
2 # Demonstrating exception arguments and stack unwinding.
3
4 import traceback
5
6 def function1():
7     function2()
8
9 def function2():
10    function3()
11
12 def function3():
13
14    # raise exception, catch exception, reraise exception
15    try:
16        raise Exception, "An exception has occurred"
17    except Exception:
18        print "Caught exception in function3. Reraising...\n"
19        raise # reraises most recent exception
20

```



```

21 # call function1, any Exception it generates will be
22 # caught by the except clause that follows
23 try:
24     function1()
25
26 # output exception arguments, string representation of exception,
27 # and the traceback
28 except Exception, exception:
29     print "Exception caught in main program."
30     print "\nException arguments:", exception.args
31     print "\nException message:", exception
32     print "\nTraceback:"
33     traceback.print_exc()

```

```

Caught exception in function3. Reraising....

Exception caught in main program.

Exception arguments: ('An exception has occurred',)

Exception message: An exception has occurred

Traceback:
Traceback (most recent call last):
  File "fig12_04.py", line 24, in ?
    function1()
  File "fig12_04.py", line 7, in function1
    function2()
  File "fig12_04.py", line 10, in function2
    function3()
  File "fig12_04.py", line 16, in function3
    raise Exception, "An exception has occurred"
Exception: An exception has occurred

```

图 12.4 异常参数和堆栈辗转开解

解释器从第1行开始执行程序。从技术角度说，这是主程序的第一行。主程序是函数调用堆栈的第一个入口，因为它是负责调用其他所有函数的实体。在主程序中，try suite 的第24行调用函数 function1（定义于第6~7行），该函数成为堆栈上的第二个入口。如 function1 引发一个异常，第28~33行的 except 处理程序便会捕捉异常，并输出与这个异常有关的信息。function1 的第7行调用 function2（定义于第9~10行），后者成为堆栈上的第3个入口。然后，function2 的第10行调用 function3（定义于第12~19行），后者成为堆栈上的第4个入口。

在这个时刻，程序的调用堆栈如下：

```

function3 (顶部)
function2
function1
主程序

```

换言之，最后一个调用的函数（function3）位于顶部，而主程序位于底部。function3 的第16行引发一个 Exception，并将 "An exception has occurred" 作为参数传递。为响应 raise 语句，Python 使用指定的参数，创建一个 Exception 对象。第17~19行的 except 子句捕捉异常，并第一个打印出消息。第19行使用一个空的 raise 语句“重新引发”异常。重新引发一个异常，通常意味着 except 处理程序只对异常执行了局部处理，现在要将异常传回调用者（目前是 function2），以便进一步处理。在本例中，function3 演示了如何在指定异常名称的前提下，使用关键字 raise 来重新引发最近引发过的一个异常。

软件工程知识 12.10 假如一个函数能处理特定类型的异常，就放手让函数处理它，不要将异常传到程序的另一个区域。

接着，function3 终止，因为重新引发的异常未在函数主体中捕捉。所以，控制权会返回当初调用 function3 的语句（或者调用堆栈中的上一个函数，即 function2）。这样会将 function3 从函数调用堆栈中

删除（或称“辗转开解”），造成函数终止。与此同时，Python 会创建一个 traceback 对象，用它维护来有关函数调用的信息。

控制权返回 function2 的第 10 行后，解释器查明第 10 行不在一个 try suite 中。所以，异常无法在 function2 中捕捉，这导致 function2 终止，而且 function2 也从函数调用堆栈中“辗转开解”，并创建另一个 traceback 对象（代表当前的辗转开解级别），并将控制权返回 function1 中的第 7 行。同样的事情再度发生。第 7 行不在一个 try suite 中，异常无法在 function1 中捕捉。这导致函数终止，并从调用堆栈中“辗转开解”。与此同时，创建另一个 traceback 对象，并使控制权返回至主程序的第 24 行。由于第 24 行在一个 try suite 中，所以主程序的 try suite 会立即“过期”，由第 28~33 行的 except 处理程序捕捉异常。

注意，第 28 行的 except 子句有别于前面所示的任何 except 子句。对 except 子句来说，如果关键字 except 之后跟一个异常类型（或由多个异常类型构成的一个元组），一个逗号，以及一个标识符，Python 就会将标识符与相匹配的异常对象绑定。以后，except 处理程序就可使用标识符获取与发生的特定异常有关的信息。第 29~33 行的 except suite 打印异常对象的 args 属性（第 30 行）。然后，处理程序打印异常的字符串表示。至于 Python 如何用字符串来表示一个异常对象，具体取决于它的 args 属性值。如果 args 属性是一个空元组，Python 就将异常表示成一个空字符串。如果异常对象的 args 元组只包含一个值，Python 就将异常表示成那个值的字符串表示。如果异常对象的 args 元组包含多个项目，Python 就将异常表示成 args 元组的字符串表示。在本例中，由于异常对象的 args 属性只包含一个值，所以 Python 将异常表示成那个值，即字符串 "An exception has occurred"。

except 处理程序的第 33 行调用函数 traceback.print_exc，从而打印出这个 traceback。traceback 模块包含了许多用于处理 traceback 对象的函数，这些对象是在堆栈辗转开解期间由 Python 创建的。记住堆栈辗转开解会一直继续下去，直到一个 except 处理程序捕捉到异常，或者程序终止。如果调用 print_exc 函数时未指定任何参数，会打印堆栈辗转开解过程中当前所积累的所有 traceback 对象。这个输出完全等同于当解释器遇到一个未捕捉的异常时由 Python 生成的输出。让我们具体探讨一下 print_exc 函数的输出。第 1 行：

```
Traceback (most recent call last)
```

是发生错误时，由 Python 打印的标准 traceback 行。这一行表明，最近的一次调用（即发生异常时位于调用堆栈顶部的那个调用）在 traceback 的输出中最后一个出现。traceback 中，第 2~3 行包含了与函数调用堆栈上的第一个调用（即从主程序中调用 function1）有关的信息。具体信息包括在其中发生调用的那个文件（fig12_04.py）、调用函数的行号（24）以及发出函数调用的实体（与主程序对应的？）。在 traceback 输出中，后续的每两行都对应于函数调用堆栈上的一个调用。倒数第二行包含了导致异常的代码（即 function3 中的第 16 行，其中包含 raise 语句）。这便证明了一个事实：第 19 行的空 raise 语句只是重新引发了第 16 行的异常。输出的最后一行包含异常类型及其参数的一个字符串表示。注意在 traceback 输出中，与调用堆栈有关的信息是从异常发生的位置开始，一直到异常被捕捉的位置（如果未被捕捉，则是程序终止的位置）。

测试和调试提示 12.11 阅读 traceback 报告时，请按从下到上顺序，先读错误消息。然后，向上阅读 trace 剩余的部分，查找报告中的第一个行号。通常，这就是导致异常的位置。

12.8 程序自定义异常类

程序员通常可用 Python 层次结构中现有的异常类来标明程序中发生的异常。但在某些情况下，也可新建自己的异常类型，使其与程序中发生的问题严格对应。这种“程序员自定义异常类”应直接或间接继承于 Exception 类。

良好编程习惯 12.1 使每类问题都与一个命名得体的异常类关联，使程序结构清晰易读。

良好编程习惯 12.2 创建程序员自定义异常类之前，应分析 Python 层次结构中现有的异常类，看是否有可满足自己需要的类。

良好编程习惯 12.3 只有在程序需要捕捉和处理与现有异常类型不同的新异常时，才定义新的异常类。

图 12.5 演示了如何定义和使用程序员自定义异常类。NegativeNumberError 类（第 6~8 行）是一个程序员自定义异常类，代表程序对负数执行非法运算时发生的异常（比如求负数的平方根）。

```

1 # Fig. 12.5: fig12_05.py
2 # Demonstrating a programmer-defined exception class.
3
4 import math
5
6 class NegativeNumberError( ArithmeticError ):
7     """Attempted improper operation on negative number."""
8     pass
9
10 def squareRoot( number ):
11     """Computes square root of number. Raises NegativeNumberError
12     if number is less than 0."""
13
14     if number < 0:
15         raise NegativeNumberError, \
16             "Square root of negative number not permitted"
17
18     return math.sqrt( number )
19
20 while 1:
21
22     # get user-entered number and compute square root
23     try:
24         userValue = float( raw_input( "\nPlease enter a number: " ) )
25         print squareRoot( userValue )
26
27     # float raises ValueError if input is not numerical
28     except ValueError:
29         print "The entered value is not a number"
30
31     # squareRoot raises NegativeNumberError if number is negative
32     except NegativeNumberError, exception:
33         print exception
34
35     # successful execution: terminate while loop
36     else:
37         break

```

```

Please enter a number: hello
The entered value is not a number

Please enter a number: -900
Square root of negative number not permitted

Please enter a number: 12.345
3.51354521815

```

图 12.5 程序员自定义异常类

Python 异常类层次结构中定义了多个异常类别，程序员自定义异常应对其中一个类别中的对应异常进行扩展。NegativeNumberError 异常通常在算术运算期间发生，所以合理的做法是从 ArithmeticError 类派生 NegativeNumberError。在 Python 中可轻松地创建一个简单的程序员自定义异常，因为新异常类的所有功能都可从基类异常继承。所以，类的主体只包含关键字 pass——代表一个不做任何事情的空套或代码块。

程序剩余的部分（第 10~37 行）演示了这个程序员自定义异常类。程序允许用户输入一个数值，然后调用函数 squareRoot（第 10~18 行）以计算那个值的平方根。为此，squareRoot 需要调用函数

`math.sqrt`，后者要求用一个非负的值作为自己的参数。如果 `math.sqrt` 收到的是一个负值，函数会引发一个 `ValueError` 异常，并传递参数“`math domain error`”。在这个程序中，我们的目的是写一个自己的平方根函数，它通过程序员自定义异常防止用户计算一个负数的平方根。如果从用户那里收到的数值是负的，`squareRoot` 会引发一个 `NegativeNumberError` 异常（第 14~16 行）。否则，`squareRoot` 就调用函数 `math.sqrt` 来计算平方根。

在主程序中，除非用户输入一个非负的值，否则 `while` 循环（第 20~37 行）不会终止。`try suite`（第 24~25 行）尝试从用户处获取一个数值，并将那个值传给 `squareRoot` 函数。用户输入数值并按回车键后，程序便将输入的值传给 `float` 函数。如果值不是一个数字，`float` 函数会引发 `ValueError` 异常，第 28~29 行的异常处理程序会打印一条错误消息。然后，控制权返回 `while` 循环的起始处。如果用户输入的是负数，`squareRoot` 函数就引发一个 `NegativeNumberError` 异常。第 32~33 行的异常处理程序打印异常对象，然后让控制权回到 `while` 循环的起始处。如果用户输入有效的非负数字，第 25 行将打印这个数字的平方根，并使程序控制进入第 36~37 行的 `else` 子句。`else suite` 只包含关键字 `break`，从而终止 `while` 循环。

本章展示了异常处理机制的工作原理，并讨论了如何编写异常处理程序来处理潜在问题，从而增强应用程序的可靠性。开发新应用程序时，必须了解程序调用的函数或者解释器可能引发哪些潜在的异常，再实现相应的异常处理代码，使应用程序更可靠。第 13 章将首先讨论软件开发的一系列实用技术。将这些技术与严格的异常处理结合使用，可创建出实用的、可靠的、有价值的软件组件。

第 13 章 字符串处理和正则表达式

学习目标

- 理解 Python 中的文本处理
- 使用 Python 的字符串数据类型方法
- 处理和搜索字符串内容
- 理解和创建正则表达式
- 使用正则表达式在字符串中匹配模式
- 使用元字符、特殊序列和分组来创建复杂正则表达式

13.1 概述

本章介绍 Python 的字符串和字符处理功能，并演示如何用正则表达式在文本中搜索模式。利用本章的技术，可开发文本编辑器、字处理程序、页面布局软件、计算机排版系统以及其他文字处理软件。以前各章已经展示了一些字处理功能。本章要扩充这方面的信息，详细介绍基本字符串数据类型提供的各种方法，以及 Python 的 re 模块所提供的强大文字处理能力。

13.2 字符和字符串基础

字符（数位、字母和符号，比如 \$、@、% 和 *）是 Python 程序的基本构件。每个程序都由字符构成。这些字符按照有意义的方式组合，表示解释器用于执行一项任务的一系列指令。每个字符都有对应的“字符代码”（有时称为“整数序数值”）。例如，整数值 122 对应于字符常量“z”。Python 提供了 ord 函数，它取得一个字符作为参数，并返回该字符的字符代码（参见图 13.1 的交互式会话）。在大多数现代程序语言和系统中，字符值是根据“Unicode 字符集”建立的。Unicode 是一种国际性字符集，其中包含的符号和字母比 ASCII 字符集多得多。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> ord("z")
122
>>> ord("\n")
10
```

图 13.1 字符的整数序数值

Python 中的字符串是一种基本数据类型。记住，字符串是一种“不可变序列”，创建之后就不能改变。前面介绍了如何使用 len 函数获取一个字符串的长度，如何用运算符+连接字符串，以及如何用运算符%来格式化字符串。字符串还支持各种方法，它们可执行其他各种格式化及处理功能。图 13.2 的表格总结了这些字符串方法。程序调用一个字符串方法来修改字符串时，方法实际会将结果作为一个新字符串返回。表中提到的“原始字符串”是指对其调用方法的那个字符串。后文将深入讨论其中的许多方法。

字符串方法	说明
capitalize()	返回原始字符串的首字母大写版本。将其他任何大写字母都转换成小写
center(width)	返回宽度为 width 的一个字符串，并让原始字符串在其中居中（两边用空格填充）
count(substring[, start[, end]])	返回 substring 在原始字符串中出现的次数。如果指定了 start 参数，就从这个索引位置开始搜索。如果还指定了 end 参数，就从 start 开始搜索，在 end 停止
encode([encoding[, errors]])	返回一个编码的字符串。Python 的默认编码方式（encoding）是标准 ASCII。errors 参

字符串方法	说明
	数定义要使用的错误处理类型，默认为"strict"
<code>endswith(substring[, start[, end]])</code>	如果字符串以 <i>substring</i> 结束，就返回 1；否则返回 0。如果指定了 <i>start</i> 参数，就在那个索引位置开始搜索。如果还指定了 <i>end</i> 参数，方法就在 <i>start:end</i> 这个“分片”中搜索
<code>expandtabs([tabsize])</code>	返回一个新字符串，其中所有制表符都被替换成空格。可选的 <i>tabsize</i> 参数指定了用于替代一个制表符的空格字符数，默认为 8
<code>find(substring[, start[, end]])</code>	返回 <i>substring</i> 在字符串中出现时的最低索引位置；如果字符串不包括该 <i>substring</i> ，就返回 -1。如果指定了 <i>start</i> 参数，就在那个索引位置开始搜索。如果还指定了 <i>end</i> 参数，方法就在 <i>start:end</i> 这个分片中搜索
<code>index(substring[, start[, end]])</code>	搜索与 <code>find</code> 方法相同的操作，但假如在字符串中没有发现 <i>substring</i> ，就引发一个 <code>ValueError</code> 异常
<code>isalnum()</code>	如果字符串只包含字母/数字字符，就返回 1；否则返回 0
<code>isalpha()</code>	如果字符串只包含字母，就返回 1；否则返回 0
<code>isdigit()</code>	如果字符串只包含数字字符（比如"0"，"1"和"2"），就返回 1；否则返回 0
<code>islower()</code>	如果字符串中的所有字母字符都是小写（比如"a"，"b"和"c"），就返回 1；否则返回 0
<code>isspace()</code>	如果字符串只包含空白字符，就返回 1；否则返回 0
<code>istitle()</code>	如果字符串中各单词的第一个字符是该单词中惟一的大写字母，就返回 1；否则返回 0
<code>isupper()</code>	如果字符串中的所有字母字符都是大写（比如"A"，"B"和"C"），就返回 1；否则返回 0
<code>join(sequence)</code>	返回一个字符串，它连接了 <i>sequence</i> （序列）中的所有字符串，并将原始字符串作为各个被连接字符串的分隔（定界）符使用
<code>ljust(width)</code>	返回一个新字符串，原始字符串在宽度为 <i>width</i> 的一个空白字符串中左对齐
<code>lower()</code>	返回一个新字符串，将原始字符串中的所有字符都转变成小写形式
<code>lstrip()</code>	返回一个新字符串，删除开头的所有空白字符
<code>replace(old, new[, maximum])</code>	返回一个新字符串，将原始字符串中出现的所有 <i>old</i> 都替换成 <i>new</i> 。可选的 <i>maximum</i> 参数指定最多要执行几次替换
<code>rfind(substring[, start[, end]])</code>	返回 <i>substring</i> 在字符串中出现的最高索引位置；如果字符串不包括该 <i>substring</i> ，就返回 -1。如果指定了 <i>start</i> 参数，就在那个索引位置开始搜索。如果还指定了 <i>end</i> 参数，方法就在 <i>start:end</i> 这个分片中搜索
<code>rindex(substring[, start[, end]])</code>	执行与 <code>rfind</code> 方法相同的操作，只是在字符串不包含 <i>substring</i> 的前提下引发一个 <code>ValueError</code> 异常
<code>rjust(width)</code>	返回一个新字符串，原始字符串在宽度为 <i>width</i> 的一个空白字符串中右对齐
<code>rstrip()</code>	返回一个新字符串，删除末尾的所有空白字符
<code>split([separator[, maximum]])</code>	返回一个由于字符串构成的列表，它在每个 <i>separator</i> （分隔符）处对原始字符串进行分解。如果省略可选的 <i>separator</i> 参数，或者设为 <code>None</code> ，会在任何空白字符序列处对原始字符串进行分解——这相当于返回一个单词列表。 <i>maximum</i> 参数规定最多进行多少次分解
<code>splitlines([keepbreaks])</code>	返回一个由字符串构成的列表，它在每个换行符（newline）处分解原始字符串。如果可选的 <i>keepbreaks</i> 参数为 1，在返回的列表中，子字符串会保留换行符
<code>startswith(substring[, start[, end]])</code>	如果字符串以 <i>substring</i> 开头，就返回 1；否则返回 0。如果指定了 <i>start</i> 参数，就在那个索引位置开始搜索。如果还指定了 <i>end</i> 参数，方法会在 <i>start:end</i> 分片中搜索。
<code>strip()</code>	返回一个新字符串，其中删除了原始字符串开头和结尾的所有空白字符
<code>swapcase()</code>	返回一个新的字符串，将原始字符串的所有大写字母转换成小写，所有小写字母转换成大写
<code>title()</code>	返回一个新字符串，使每个单词的首字母大写，单词中的其他字母小写

字符串方法	说明
<code>translate(table[, delete])</code>	将原始字符串转换成一个新字符串。首先删除可选参数 <i>delete</i> 中的所有字符，然后将原始字符串的每个字符 <i>c</i> 替换成 <code>table[ord(c)]</code> 值
<code>upper()</code>	返回一个新字符串，将原始字符串中的所有字符都转换成大写形式

图 13.2 字符串方法

13.3 字符串表示

有许多原因要求对字符串进行格式化。例如，通过操纵字符串的表示，用户可以更容易地阅读和理解程序指令或者输出。本节用两个简单的例子来演示字符串格式化方法。图 13.3 使用了 3 个字符串方法，即 `center`、`ljust` 和 `rjust`，它们用于对齐字符串。这些方法用空白字符来操纵字符串的格式化。

```
1 # Fig. 13.3: fig13_03.py
2 # Simple output formatting example.
3
4 string1 = "Now I am here."
5
6 print string1.center( 50 )
7 print string1.rjust( 50 )
8 print string1.ljust( 50 )
```

```
Now I am here.
Now I am here.
Now I am here.
```

图 13.3 字符串的对齐方式

第 6 行的字符串方法 `center` 取得一个参数（整数值），它指定了最终输出的字符串的总长度（宽度）。然后，方法新建这个长度的一个字符串，并让原始字符串（`string1`）居中，使其左右两侧都有相同数量的空格数。字符串方法 `rjust` 则在原始字符串的左侧添加 `50 - len(string1)` 个空格字符，使字符串右对齐（第 7 行）。第 8 行使用 `ljust` 方法使字符串左对齐，它在原始字符串的右侧添加 `50 - len(string1)` 个空格字符。假如原始字符串的长度超出了这些方法的参数所指定的值，方法就会返回原始字符串。

图 13.4 演示如何剔除空白字符。第 4 行创建字符串 `string1`，它的前后都有空白字符。字符串方法 `strip` 从原始字符串删除这些空白（第 7 行）。`lstrip` 方法只删除前面的空白（第 8 行），`rstrip` 只删除后面的空白（第 9 行）。如输出结果所示，这些方法会删除所有空白字符，其中包括空格字符、换行符以及制表符。

```
1 # Fig. 13.4: fig13_04.py
2 # Stripping whitespace from a string.
3
4 string1 = "\t \n This is a test string. \t\t \n"
5
6 print 'Original string: "%s"\n' % string1
7 print 'Using strip: "%s"\n' % string1.strip()
8 print 'Using left strip: "%s"\n' % string1.lstrip()
9 print "Using right strip: \"%s\"\n" % string1.rstrip()
```

```
Original string: "
  This is a test string.
"
Using strip: "This is a test string."
Using left strip: "This is a test string.
"
Using right strip: "
  This is a test string."
```

图 13.4 从字符串中剔除空白

13.4 搜索字符串

在许多应用程序中，都有必要在一个字符串中搜索字符或字符集。例如，程序员可能希望创建一个字处理程序，并提供文字查找功能。为执行这种任务，Python 提供了 `find` 和 `index` 等方法。搜索一个子字符串时，既可判断字符串中是否包含该子字符串，也可获取该子字符串的起始索引。图 13.5 演示了如何在字符串的开头、中间和尾部搜索子字符串。

```

1 # Fig. 13.5: fig13_05.py
2 # Searching strings for a substring.
3
4 # counting the occurrences of a substring
5 string1 = "Test1, test2, test3, test4, Test5, test6"
6
7 print '"test" occurs %d times in \n\t%s' % \
8       ( string1.count( "test" ), string1 )
9 print '"test" occurs %d times after 18th character in \n\t%s' % \
10      ( string1.count( "test", 18, len( string1 ) ), string1 )
11 print
12
13 # finding a substring in a string
14 string2 = "Odd or even"
15
16 print '"%s" contains "or" starting at index %d' % \
17       ( string2, string2.find( "or" ) )
18
19 # find index of "even"
20 try:
21     print '"even" index is', string2.index( "even" )
22 except ValueError:
23     print '"even" does not occur in "%s"' % string2
24
25 if string2.startswith( "Odd" ):
26     print '"%s" starts with "Odd"' % string2
27
28 if string2.endswith( "even" ):
29     print '"%s" ends with "even"\n' % string2
30
31 # searching from end of string
32 print 'Index from end of "test" in "%s" is %d' \
33       % ( string1, string1.rfind( "test" ) )
34 print
35
36 # find rindex of "Test"
37 try:
38     print 'First occurrence of "Test" from end at index', \
39           string1.rindex( "Test" )
40 except ValueError:
41     print '"Test" does not occur in "%s"' % string1
42
43 print
44
45 # replacing a substring
46 string3 = "One, one, one, one, one, one"
47
48 print "Original:", string3
49 print 'Replaced "one" with "two":', \
50       string3.replace( "one", "two" )
51 print "Replaced 3 maximum:", string3.replace( "one", "two", 3 )

```

```

"test" occurs 4 times in
Test1, test2, test3, test4, Test5, test6
"test" occurs 2 times after 18th character in
Test1, test2, test3, test4, Test5, test6

"Odd or even" contains "or" starting at index 4
"even" index is 7

```



```

"Odd or even" starts with "Odd"
"Odd or even" ends with "even"

Index from end of "test" in "Test1, test2, test3, test4, Test5, test6" is 35

First occurrence of "Test" from end at index 28

Original: One, one, one, one, one, one
Replaced "one" with "two": One, two, two, two, two, two
Replaced 3 maximum: One, two, two, two, one, one

```

图 13.5 在字符串中搜索子字符串

第 5~11 行使用字符串方法 `count` 返回子字符串在字符串或者字符串分片中出现的次数。如果方法没有找到指定的子字符串，方法会返回 0。第 8 行打印子字符串“test”在 `string1` 中出现的次数。`count` 方法可指定两个可选参数，指定要在字符串的哪个分片中搜索。第 10 行为 `count` 传递参数，要求方法从索引位置 18（即字符“3”）开始搜索，并一直搜索到字符串末尾。这个调用的效果相当于执行以下语句：

```
string1[18:len(string1)].count("test")
```

但通过为方法调用提供可选参数，既能提高可读性，还能改进程序性能，因为不必另外新建一个分片。

第 14~29 行演示了在字符串中搜索子字符串的过程。第 17 行使用 `find` 方法返回子字符串出现时的最低索引位置。如果字符串中不包含指定的子字符串，方法返回 -1。`index` 方法（第 21 行）类似于 `find` 方法，只是假如字符串中不包含子字符串，方法会引发 `ValueError` 异常。在这种情况下，程序可以捕捉该异常，并进行相应的处理。

第 25~29 行使用了一个方法，用于判断一个字符串是否以指定子字符串开始或结尾。如果字符串以指定的子字符串开始，`startswith` 方法就返回 1（第 25 行）。这个调用的效果等价于以下表达式：

```
string2[0:len("Odd")] == "Odd"
```

如果字符串以指定子字符串结尾，`endswith` 方法就返回 1（第 28 行）。效果等价于表达式：

```
string2[-len("even"):] == "even"
```

程序可从字符串的尾部开始搜索一个子字符串。第 32~43 行用 `rfind` 和 `rindex` 方法判断 `string1` 是否包含特定的子字符串。`rfind` 方法从字符串尾部开始搜索，返回首次出现于字符串的索引位置。如果没有发现子字符串，方法返回 -1。`rindex` 方法则返回子字符串出现的最高索引位置，没有发现子串就引发 `ValueError` 异常。我们的程序捕捉这个异常，以处理字符串中不包含指定的子字符串的情况。

用户偶尔需要在找到一个子字符串后，再对其执行特定的操作。例如，可能希望在文档中查找一句话，然后将其替换成另一句话。`replace` 方法可取得两个子字符串，并在文档中搜索第一个子字符串，把它替换成第二个子字符串。第 50 行将 `string3` 中出现的所有“one”都替换成“two”。`replace` 方法还可取得第 3 个参数，它指定了最大的替换次数。第 51 行指定将“one”替换成“two”，最多只执行 3 次。

13.5 连接和分解字符串

计算机处理代码的方式与我们阅读文字的方式相似。我们读一个句子时，大脑自然就会将它分解成单独的“字或单词”（或称“标记”），每个都能表达特定含义。这个过程称为“标记化”（Tokenization）。Python 解释器也要执行标记化操作，因为它会将程序语句分解成单独的程序元素，比如关键字、标识符、运算符等等。不同的标记用定界符分隔，定界符通常是空白字符，比如空格、制表符、换行符和回车。当然，也可用其他字符来分隔标记。本节介绍的方法可进行基于定界符的字符串分解与连接。

图 13.6 演示了字符串方法 `split` 和 `join`。第 5 行创建 `string1`，其中包含用逗号分隔的一系列字母。第 7~11 行展示如何根据定界符来分解字符串。第 8 行调用无参数的 `split` 方法，它会在每个空白字符位置

分解字符串。该方法返回一系列“标记”，并由程序显示出来。在第 9 行，`split` 接收参数`","`，表明定界符是逗号，所以会在每个逗号位置分解字符串。在第 10 行，`split` 方法接收两个参数——定界符和指定最大分解次数的一个整数。

```

1 # Fig. 13.6: fig13_06.py
2 # Token splitting and delimiter joining.
3
4 # splitting strings
5 string1 = "A, B, C, D, E, F"
6
7 print "String is:", string1
8 print "Split string by spaces:", string1.split()
9 print "Split string by commas:", string1.split(",")
10 print "Split string by commas, max 2:", string1.split(",", 2)
11 print
12
13 # joining strings
14 list1 = [ "A", "B", "C", "D", "E", "F" ]
15 string2 = "___"
16
17 print "List is:", list1
18 print 'Joining with "%s": %s' \
19       % ( string2, string2.join( list1 ) )
20 print 'Joining with "-.-":', "-.-".join( list1 )

```

```

String is: A, B, C, D, E, F
Split string by spaces: ['A', 'B', 'C', 'D', 'E', 'F']
Split string by commas: ['A', 'B', 'C', 'D', 'E', 'F']
Split string by commas, max 2: ['A', 'B', 'C, D, E, F']

List is: ['A', 'B', 'C', 'D', 'E', 'F']
Joining with "___": A___B___C___D___E___F
Joining with "-.-": A-.-B-.-C-.-D-.-E-.-F

```

图 13.6 分解和连接字符串

针对一个标记列表，`join` 方法可用预定义的定界符来合并列表。第 14 行创建由字母标记构成的一个列表，第 15 行创建定界符 `string2`，其中包括 3 个下划线（`"_"`）字符。第 18~19 行显示了调用 `string2` 的 `join` 方法的结果。该方法取得一个标记列表作为参数，并返回在 `string2` 中用下划线定界符进行连接的结果。第 20 行演示了如何将 `print` 方法将字符串的 `join` 方法调用合并起来。

性能提示 13.1 构建复杂字符串时，效率更高的一种做法是将不同组件包括到一个列表中，再用 `join` 方法汇总字符串，而不要使用连接运算符（`+`）。

13.6 正则表达式

字符串方法允许程序查找一个特定的子字符串。例如，要想判断字符串中是否包含一个代表星期几的字符串（`"Monday"`，`"Tuesday"`，`"Wednesday"` 等等），程序可为每个子字符串都调用字符串方法 `find`。换言之，程序要调用 7 次 `find` 方法，才能检查完从周一到周日的所有可能。对于更复杂的搜索，程序则可能要调用许多次 `find` 方法。显然，这并不是一种高效的解决方案。为此，“正则表达式”（`Regular Expression`）提供了更高效、功能更强的解决方案。所谓正则表达式，实际是一个“文本模式”（`Text Pattern`），用于查找与模式相匹配的子字符串。本章后文将讨论 Python 的各种正则表达式功能。

良好编程习惯 13.1 如果进行的是简单的处理，请使用字符串方法，这样可避免复杂的正则表达式所带来的错误，并保证程序的可读性。

先来看一个简单的例子（图 13.7）。我们要搜索各种欢迎词：`"hello"`，`"Hello"` 和 `"world!"`。

```

1 # Fig. 13.7: fig13_07.py
2 # Simple regular-expression example.
3
4 import re
5
6 # list of strings to search and expressions used to search
7 testStrings = [ "Hello World", "Hello world!", "hello world" ]
8 expressions = [ "hello", "Hello", "world!" ]
9
10 # search every expression in every string
11 for string in testStrings:
12
13     for expression in expressions:
14
15         if re.search( expression, string ):
16             print expression, "found in string", string
17         else:
18             print expression, "not found in string", string
19
20     print

```

```

hello not found in string Hello World
Hello found in string Hello World
world! not found in string Hello World

hello not found in string Hello world!
Hello found in string Hello world!
world! found in string Hello world!

hello found in string hello world
Hello not found in string hello world
world! not found in string hello world

```

图 13.7 正则表达式示例

第 4 行导入正则表达式模块 `re`，它为 Python 提供了正则表达式处理能力。列表 `testStrings`（第 7 行）包含几个要用第 8 行创建的正则表达式搜索的字符串。注意，正则表达式和普通字符串非常相似。

程序剩余部分包括一个嵌套 `for` 循环，它会检查列表 `testStrings` 中的每个字符串，判断其中是否包含列表 `expressions` 中的任何一个正则表达式。`re.search` 函数在字符串中查找第一次出现的正则表达式，并返回一个对象，该对象中包含与正则表达式匹配的子字符串。如果字符串不包含指定模式，`re.search` 就返回 `None`。程序确定函数调用是否返回一个值，然后打印相应的消息。下一节将讨论如何使用 `re.search` 返回的值。

本例的每个正则表达式都是某个测试字符串的一个子串。事实上，第 15 行可替换成以下表达式：

```
if string.find( expression ) >= 0:
```

而且程序会生成相同结果。后文将探讨如何创建功能更强的正则表达式模式字符串。

13.7 编译正则表达式和处理正则表达式对象

本节要讨论“编译好的正则表达式对象”，并介绍 `re.search` 返回的对象（其中包含搜索结果）。通常情况下，`re` 模块会将正则表达式编译成特定形式，以便模块将其用于处理一个字符串。如果程序需要多次使用相同的正则表达式，请提前编译好正则表达式，以提高程序效率。图 13.8 演示了如何提前编译正则表达式，创建编译好的正则表达式对象。另外，这个例子还展示了如何利用 `re.search` 返回的对象来查看搜索结果。

```

1 # Fig. 13.08: fig13_08.py
2 # Compiled regular-expression and match objects.
3
4 import re

```

```

5
6 testString = "Hello world"
7 formatString = "%-35s: %s" # string for formatting the output
8
9 # create regular expression and compiled expression
10 expression = "Hello"
11 compiledExpression = re.compile( expression )
12
13 # print expression and compiled expression
14 print formatString % ( "The expression", expression )
15 print formatString % ( "The compiled expression",
16     compiledExpression )
17
18 # search using re.search and compiled expression's search method
19 print formatString % ( "Non-compiled search",
20     re.search( expression, testString ) )
21 print formatString % ( "Compiled search",
22     compiledExpression.search( testString ) )
23
24 # print results of searching
25 print formatString % ( "search SRE_Match contains",
26     re.search( expression, testString ).group() )
27 print formatString % ( "compiled search SRE_Match contains",
28     compiledExpression.search( testString ).group() )

```

```

The expression           : Hello
The compiled expression  : <SRE_Pattern object at 0x00B60A20>
Non-compiled search      : <SRE_Match object at 0x00D0F9B8>
Compiled search          : <SRE_Match object at 0x00D0F9B8>
search SRE_Match contains : Hello
compiled search SRE_Match contains : Hello

```

图 13.8 编译正则表达式

第 11 行的 `re.compile` 函数取得一个正则表达式作为参数，并返回一个 `SRE_Pattern` 对象，它代表编译好的正则表达式。编译好的正则表达式对象具有 `re` 模块的全部功能。例如，`compiledExpression` 的 `search` 方法（第 22 行）对应于函数 `re.search`（第 20 行）。如输出结果所示，这两种方式都能返回一个 `SRE_Match` 对象。该对象提供了多个用于获取正则表达式处理结果的方法。`group` 方法（第 26~28 行）返回与模式匹配的子字符串。13.11 节在讨论分组问题时，将进一步解释这个方法。

13.8 正则表达式的重复和置位字符

现在来讨论如何构建更复杂的模式字符串。正则表达式就像一种“语言中的语言”。就像 Python 为程序创建定义了一套严格的语法元素那样，正则表达式也规定了几个用于创建模式的特殊字符。大多数模式都是由字符、元字符和转义序列等基本元素组合而成的。所谓“元字符”（Metacharacters），实际是正则表达式的一种语法元素（就像关键字 `if` 是一种 Python 语法元素那样）。元字符对其本身进行匹配。元字符的任务是对一个或多个字符进行重复、分组、置位或分类。本节要介绍用于重复的元字符。以后的小节将讨论转义序列和其他元字符。

图 13.9 演示了基本的重复元字符 `?`、`+` 和 `*`。第 7 行创建一个正则表达式列表，该列表中包含这些符号。元字符 `?` 匹配零个或一个在它之前的表达式。表达式可为单个字符、一个转义序列、一个字符类（详情参见 13.9 节）或者一个组（详情参见 13.11 节）。在这个简单的例子中，我们只匹配单个字符。例如，第 7 行的第一个表达式是 `"Hel?o"`，它匹配的模式由一个字母 `H`、字母 `e`、零或多个字母 `l` 及字母 `o` 组成。

```

1 # Fig. 13.9: fig13_09.py
2 # Repetition patterns, matching vs searching.
3
4 import re
5
6 testStrings = [ "Heo", "Helo", "Helllllo" ]
7 expressions = [ "Hel?o", "Hel+o", "Hel*o" ]

```



```

8
9 # match every expression with every string
10 for expression in expressions:
11
12     for string in testStrings:
13
14         if re.match( expression, string ):
15             print expression, "matches", string
16         else:
17             print expression, "does not match", string
18
19     print
20
21 # demonstrate the difference between matching and searching
22 expression1 = "elo" # plain string
23 expression2 = "^elo" # "elo" at beginning of string
24 expression3 = "elo$" # "elo" at end of string
25
26 # match expression1 with testStrings[ 1 ]
27 if re.match( expression1, testStrings[ 1 ] ):
28     print expression1, "matches", testStrings[ 1 ]
29
30 # search for expression1 in testStrings[ 1 ]
31 if re.search( expression1, testStrings[ 1 ] ):
32     print expression1, "found in", testStrings[ 1 ]
33
34 # search for expression2 in testStrings[ 1 ]
35 if re.search( expression2, testStrings[ 1 ] ):
36     print expression2, "found in", testStrings[ 1 ]
37
38 # search for expression3 in testStrings[ 1 ]
39 if re.search( expression3, testStrings[ 1 ] ):
40     print expression3, "found in", testStrings[ 1 ]

```

```

Hel?o matches Heo
Hel?o matches Helo
Hel?o does not match Hellllo

Hel+o does not match Heo
Hel+o matches Helo
Hel+o matches Hellllo

Hel*o matches Heo
Hel*o matches Helo
Hel*o matches Hellllo

elo found in Helo
elo$ found in Helo

```

图 13.9 用重复元字符搜索和匹配字符串

元字符+匹配的是在它之前的表达式的一次或多次出现。例如，第 7 行的第二个正则表达式是“Hel+o”，它匹配的模式由字母 H、字母 e、一次或多次出现的字母 l 及字母 o 组成。元字符*则匹配在它之前的表达式的零次或多次出现。例如，第 7 行的第 3 个正则表达式是“Hel*o”，它匹配的模式由字母 H、字母 e、零个或多个字母 l 及字母 o 组成。

第 10~19 行包含一个嵌套 for 循环，它将第 7 行的每个正则表达式应用于第 6 行的每一个字符串。re.match 函数（第 14 行）将一个表达式与一个字符串匹配。前面介绍的 re.search 函数会在字符串的任意部分和表达式匹配时返回一个 SRE_Match 对象。与之不同的是，re.match 函数会在字符串的开头与正则表达式匹配的前提下返回一个 SRE_Match 对象。

正则表达式可包含其他两个元字符，指定模式在字符串中出现的位置。元字符^表示位于字符串开头；元字符\$表示位于字符串末尾。相应地，进行搜索或匹配时，只有在字符串开头或末尾包含指定模式的前提下，才会返回一个值。第 22~40 行创建包含这些元字符的正则表达式，并用函数 re.match 和 re.search 来处理一个字符串。

第 22~24 行的正则表达式分别指定了位于字符串任何位置、开头以及末尾的"elo"字符序列。从输出结果可知,如果向函数 `re.match` 传递参数 `expression1` 和 `testStrings[2]`,它将返回 `None`,因为正则表达式"elo"不与完整的字符串"Helo"匹配。类似地,向函数 `re.search` 传递参数 `expression2` 和 `testStrings[1]`,它会返回 `None`,因为字符串"elo"不在字符串"Helo"的开头。

13.9 字符类和特殊序列

本节要讨论两个基本的正则表达式元素:字符类和特殊序列(或转义序列)。字符类指定了要在一个字符串中匹配的一组字符。特殊序列则是常用字符类的快捷方式。

元字符`[和]`代表一个“正则表达式类”,包含一个类的正则表达式与类中的单个字符相匹配。`"[abc]"`就是一个正则表达式类,它匹配的是字母 `a`、`b` 或 `c`。类可用`-`字符指定由连续的字符构成的一个范围。例如,正则表达式`"[a-d]"`等价于`"[abcd]"`。

元字符`^`如果位于一个类的开头,会对类进行“求反”。这意味着正则表达式会匹配所有字符,但类中指定的那些字符“除外”。例如,`"[^a-c]"`匹配除 `a`、`b` 和 `c` 之外的所有字符。如图 13.10 所示的特殊序列描述了常用的字符类。

特殊序列	说明
<code>\d</code>	数位类,相当于 <code>[0-9]</code>
<code>\D</code>	对数位类求反,相当于 <code>^[^0-9]</code>
<code>\s</code>	空白字符类,相当于 <code>[\n\r\t\f\v]</code>
<code>\S</code>	对空白字符类求反,相当于 <code>^[^\n\r\t\f\v]</code>
<code>\w</code>	字母、数字类,相当于 <code>[a-zA-Z0-9_]</code>
<code>\W</code>	对字母、数字类求反,相当于 <code>^[a-zA-Z0-9_]</code>
<code>\\</code>	反斜杠 (<code>\</code>)

图 13.10 正则表达式特殊序列

图 13.11 演示了其中包含类和特殊序列的正则表达式。另外还演示了如何避免常见的正则表达式错误。第 8 行的正则表达式包含一个新的元字符,并演示了使用正则表达式时的一个重点。元字符`|`既匹配左边的正则表达式,也匹配右边的正则表达式。表达式`"[abc]"`的另一个写法就是`"a|b|c"`。所以,第 8 行的正则表达式要么匹配字符串`"2x+5y"`,要么匹配字符串`"7y-3z"`。

```

1 # Fig. 13.11: fig13_11.py
2 # Program that demonstrates classes and special sequences.
3
4 import re
5
6 # specifying character classes with [ ]
7 testStrings = [ "2x+5y", "7y-3z" ]
8 expressions = [ r"2x\+5y|7y-3z",
9                 r"[0-9]{a-zA-Z0-9_}.[0-9]{yz}",
10                 r"\d\w-\d\w" ]
11
12 # match every expression with every string
13 for expression in expressions:
14     for testString in testStrings:
15         if re.match( expression, testString ):
16             print expression, "matches", testString
17
18 # specifying character classes with special sequences
19 testString1 = "800-123-4567"
20 testString2 = "617-123-4567"
21 testString3 = "email: \t joe_doe@deitel.com"
```

```

24
25 expression1 = r"^\d{3}-\d{3}-\d{4}$"
26 expression2 = r"\w+:\s+\w+@\w+\.(com|org|net)"
27
28 # matching with character classes
29 if re.match(expression1, testString1):
30     print expression1, "matches", testString1
31
32 if re.match(expression1, testString2):
33     print expression1, "matches", testString2
34
35 if re.match(expression2, testString3):
36     print expression2, "matches", testString3

```

```

2x\+5y\7y-3z matches 2x+5y
2x\+5y\7y-3z matches 7y-3z
[0-9][a-zA-Z0-9_].[0-9][yz] matches 2x+5y
[0-9][a-zA-Z0-9_].[0-9][yz] matches 7y-3z
\d\w-\d\w matches 7y-3z
^\d{3}-\d{3}-\d{4}$ matches 800-123-4567
^\d{3}-\d{3}-\d{4}$ matches 617-123-4567
\w+:\s+\w+@\w+\.(com|org|net) matches email: joe_doe@deitel.com

```

图 13.11 使用类和特殊序列的正则表达式

注意，第 8 行的正则表达式在字符+之前，使用了转义元字符\。它匹配的是字面意义的字符+，而非重复元字符+。如果不对字符+进行转义，正则表达式就匹配一个或多个 x 字符，后跟数字字符 5（如图 13.12 所示）。

```

Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
>>> import re
>>> print re.match("2x+5y", "2x+5y")
None
>>> print re.match("2x+5y", "2x5y")
<SRE_Match object at 0x00932268>
>>> print re.match("2x+5y", "2xx5y")
<SRE_Match object at 0x00949A88>

```

图 13.12 正则表达式中的元字符\

同时还要注意，第 8 行的正则表达式是一个“原始字符串”（Raw String），也就是在字符串之前加上字符前缀 r 后创建的一个字符串。通常，如果在字符串中出现\，Python 会把它视为转义字符，并试图用正确的转义序列替换元字符\及其后续字符。但是，如果元字符\出现在一个原始字符串中，Python 不会把它解释成转义字符，而是将其视为字面意义的反斜杠字符。例如，Python 会将"\n"解释成换行符，但将 r"\n"解释成两个字符——一个反斜杠和一个 n。

常见编程错误 13.1 在原始字符串末尾使用反斜杠是语法错误。

良好编程习惯 13.2 正则表达式的模式字符串中常常包含反斜杠字符。使用原始字符串来创建模式，可避免对其中每个反斜杠进行转义的必要，使模式字符串更容易理解。

第 9~10 行创建了两个其他正则表达式。元字符. 匹配字符串中除换行符之外的任何字符。第 9 行的正则表达式匹配一个数位，后面依次跟一个字母/数字字符、除换行符之外的其他任何字符、一个数位和字母 y 或 z。第 10 行的正则表达式使用特殊序列创建类似正则表达式。该表达式匹配一个数位，后面依次跟一个字母/数字字符、字符-、一个数位和一个字母/数字字符。第 13~18 行包含一个嵌套的 for 循环，它用第 8~10 行的每个表达式去匹配第 7 行的每个字符串。

程序剩余部分创建更复杂的正则表达式。元字符{和}以另一种方式重复字符。第 25 行的表达式匹配 3 个数位（如同在花括号中指定的），后面依次跟字符-、3 个数位、另一个字符-及 4 个数位。将正则表达式放到元字符^和\$之间，可让正则表达式匹配完整的字符串。还可使用花括号元字符指定重复范围。

例如，表达式`"\d{1,3}"`匹配 1 个数位、2 个数位或 3 个数位。

第 26 行创建一个正则表达式，它匹配一个或多个字母/数字字符，后面依次跟一个冒号 (:)、一个或多个空白字符、一个或多个字母/数字字符、字符@、一个或多个字母/数字字符、字符.（注意对这个正则表达式字符进行转义的反斜杠）以及任何一个字符序列：com, org 或 net。程序剩余部分则试图用正则表达式来匹配测试字符串。

13.10 正则表达式的字符串处理函数

本章前几节讨论了基本的字符串处理方法，并指出使用正则表达式，可获得功能更强的版本。re 模块提供了基于模式的字符串处理能力，比如替换子字符串，以及根据定界符对字符串进行分解等等。图 13.13 对此进行了演示。

```
1 # Fig. 13.13: fig13_13.py
2 # Regular-expression string manipulation.
3
4 import re
5
6 testString1 = "This sentence ends in 5 stars *****"
7 testString2 = "1,2,3,4,5,6,7"
8 testString3 = "1+2x*3-y"
9 formatString = "%-34s: %s" # string to format output
10
11 print formatString % ( "Original string", testString1 )
12
13 # regular expression substitution
14 testString1 = re.sub( r"\*", r"^", testString1 )
15 print formatString % ( "^ substituted for *", testString1 )
16
17 testString1 = re.sub( r"stars", "carets", testString1 )
18 print formatString % ( "carets substituted for stars",
19     testString1 )
20
21 print formatString % ( 'Every word replaced by "word"',
22     re.sub( r"\w+", "word", testString1 ) )
23
24 print formatString % ( 'Replace first 3 digits by "digit"',
25     re.sub( r"\d", "digit", testString2, 3 ) )
26
27 # regular expression splitting
28 print formatString % ( "Splitting " + testString2,
29     re.split( r",", testString2 ) )
30
31 print formatString % ( "Splitting " + testString3,
32     re.split( r"[+*%/]", testString3 ) )
```

```
Original string           : This sentence ends in 5 stars *****
^ substituted for *       : This sentence ends in 5 stars ^^^^^
"carets" substituted for "stars" : This sentence ends in 5 carets ^^^^^
Every word replaced by "word" : word word word word word word ^^^^^
Replace first 3 digits by "digit" : digit,digit,digit,4,5,6,7
Splitting 1,2,3,4,5,6,7   : ['1', '2', '3', '4', '5', '6', '7']
Splitting 1+2x*3-y        : ['1', '2x', '3', 'y']
```

图 13.13 基于正则表达式的字符串处理

re.sub 函数（第 14 行）取得 3 个参数。第一个参数指定模式，第二个参数指定子字符串，第 3 个参数指定字符串。在第 3 个参数指定的字符串中，与模式匹配的每个子字符串都会被替换成第二个参数指定的子字符串。第 14 行用脱字号 (^) 替换 testString1 字符串中的星号 (*)。为替换星号，方法必须使用正则表达式`"\""`，因为 * 是元字符。第 21~22 行将每个单词都替换成子字符串"word"。第 24~25 行使用函数可选的第 4 个参数指定最多替换多少次。

`re.split` 函数取得两个参数。第一个参数是正则表达式，用模式来描述定界符。函数在定界符处对第二个参数进行分解，返回一个标记列表。第 28~29 行打印在逗号 (,) 处对变量 `testString2` 进行分解的结果。第 32 行调用 `re.split`，传递与 5 种算术运算符匹配的一个定界符模式。注意，这个正则表达式定义了一个类，它对字符-进行了转义，但没有对字符*进行同样的处理。这演示了正则表达式的一个容易被忽视的特性：类中出现的任何字符（用于求反的字符^和用于表示范围的字符-除外）都会被解释成字面意义的字符。所以，\$、+或*等元字符在类中无需转义。

13.11 分组

图 13.8 指出程序可用 `group` 方法从 `SRE_Match` 对象提取匹配的子字符串。这个方法来源于一种更高级的正则表达式技术，即“分组”(Grouping)。正则表达式可指定要在一个字符串中匹配的子字符串“分组”。然后，程序用正则表达式搜索或匹配一个字符串，并从匹配的分组中提取信息。图 13.14 创建了含有分组的正则表达式，并打印从这些组提取的信息。

```
1 # Fig. 13.14: fig13_14.py
2 # Program that demonstrates grouping and greedy operations.
3
4 import re
5
6 formatString1 = "%-22s: %s" # string to format output
7
8 # string that contains fields and expression to extract fields
9 testString1 = \
10 "Albert Antstein, phone: 123-4567, e-mail: albert@bug2bug.com"
11 expression1 = \
12 r"(\w+ \w+), phone: (\d{3}-\d{4}), e-mail: (\w+@\w+\.\w{3})"
13
14 print formatString1 % ( "Extract all user data",
15     re.match( expression1, testString1 ).groups() )
16 print formatString1 % ( "Extract user e-mail",
17     re.match( expression1, testString1 ).group( 3 ) )
18 print
19
20 # greedy operations and grouping
21 formatString2 = "%-38s: %s" # string to format output
22
23 # strings and patterns to find base directory in a path
24 pathString = "/books/2001/python" # file path string
25
26 expression2 = "(/.+)/" # greedy operator expression
27 print formatString1 % ( "Greedy error",
28     re.match( expression2, pathString ).group( 1 ) )
29
30 expression3 = "(/.*?)/" # non-greedy operator expression
31 print formatString1 % ( "No error, base only",
32     re.match( expression3, pathString ).group( 1 ) )
```

```
Extract all user data : ('Albert Antstein', '123-4567', 'albert@
bug2bug.com')
Extract user e-mail   : albert@bug2bug.com

Greedy error          : /books/2001
No error, base only   : /books
```

图 13.14 正则表达式分组和“贪吃”运算符

第 12 行的正则表达式描述了这些组。元字符(和)标明一个组。第一个组匹配的模式是一个单词 (`\w+`)，然后依次跟一个空格和另一个单词。第二个组匹配 3 个数位，后跟字符-，最后是 4 个数位。第三个组匹配一个或多个/数字字符，后面依次跟字符@、一个或多个字母/数字字符、字符.和 3 个字母/数字字符。这个正则表达式与变量 `testString1` 中的字符串进行匹配。以上三个组分别匹配一个人的姓名、

电话号码以及电子邮件地址。

第 14~17 行演示了分组的好处。第 15 行调用函数 `re.match`，它返回一个 `SRE_Match` 对象。该对象的 `groups` 方法返回一个子字符串列表。其中每个子字符串都对应于和正则表达式中的一个组匹配的子字符串。列表中的第一个子字符串匹配正则表达式中的第一个组，并以此类推。第 15 行的结果是程序获得这个人的姓名、电话号码和电邮地址。第 17 行调用 `SRE_Match` 的 `group` 方法，传递整数值 3 作为参数。该调用返回与正则表达式的第 3 个组匹配的子字符串，结果是从 `testString1` 中提取电邮地址子字符串。

正则表达式“分组”带来了正则表达式另一个容易被忽视的问题。元字符 `+` 和 `*` 称为“贪吃运算符”，它试图匹配尽可能多的字符。但这有时并非我们希望的行为。第 20~32 行演示了贪吃运算符的问题。第 24 行的字符串包含一个示范路径，它可能是 URL 的一部分。假定要写一个正则表达式，目的是从路径中获得根目录名称（本例是 `/books`）。第 26~28 行尝试这个操作，但由于运算符 `+` 在 `expression2` 中的贪吃行为，造成操作失败。碰到贪吃运算符时，正则表达式模块会试图匹配运算符之前的尽可能多的表达式。起初，这会导致 `expression2` 匹配整个字符串。然而，正则表达式模块必须允许匹配模式剩余的部分。在这种情况下，在包含贪吃运算符的组之后，必须紧接一个正斜杠（`/`），如同在 `expression2` 中指定的那样。所以，正则表达式模块在字符串中向后搜索，直到正则表达式模块能够保证在 `expression2` 的最初的组之后有一个正斜杠（`/`）。所以，包含贪吃运算符的组最终匹配的是 `/books/2001`，这当然是错误的。

第 30 行的正则表达式修改了贪吃运算符 `+` 的行为，以便在示范路径中正确获得根目录名称。在运算符 `+` 之后插入一个元字符 `?`，即可改变运算符 `+` 的行为。现在，当正则表达式模块使用 `expression3` 来匹配字符串时，模块每次只搜索一个字符，直到发现与组中的模式匹配的最小的一个字符串（即 `/books`）。

本章首先介绍了基本字符串处理能力，然后解释如何利用 `re` 模块提供的强大的正则表达式处理能力。下一章开始讲解文件处理，它允许程序从磁盘文件中读取信息，并将信息写入文件。执行文件处理的许多程序都要用正则表达式以及其他字符串处理能力来搜索和处理文件内容。

13.12 因特网和万维网资源

以下资源可加深和拓展您对正则表达式的认识。

etext.lib.virginia.edu/helpsheets/regex.html

本教程探讨正则表达式的常见用法、编写复杂正则表达式以及包括转义字符和锚点在内的其他主题。

www.zvon.org/other/reReference/Output

本参考描述了常见的正则表达式特殊序列。

py-howto.sourceforge.net/regex/regex.html

本教程讨论如何使用正则表达式和 `re` 模块。涉及的主题包括常见问题、修改字符串和模式匹配等等。

www.devshed.com/Server_Side/Administration/RegExp

这篇文章描述了正则表达式的常见用法。

py-howto.sourceforge.net/regex/regex.html

讨论如何使用 `re` 模块，解释了贪吃运算符，以及如何将原始字符串用作正则表达式模式字符串。

第 14 章 文件处理和序列化

学习目标

- 会创建、读取、写入及更新文件
- 熟悉顺序访问文件处理
- 理解通过 `shelve` 模块进行的随机访问文件处理
- 会指定高性能、未格式化的 I/O 操作
- 理解格式化和原始数据文件处理的差异
- 利用随机访问文件处理构建一个事务处理程序
- 序列化复杂对象以便存储

14.1 概述

变量和序列只能临时存储数据，如果局部变量“超出作用域”或程序终止，数据就会丢失。文件则相反，可长期储存大量数据（即使创建数据的程序终止）。文件维护的数据通常称为“持久数据”。计算机将文件存储在“辅助存储设备”中，比如磁盘、光盘和磁带。本章描述 Python 程序如何创建、更新及处理数据文件。还介绍了“顺序访问文件”和“随机访问文件”，并分别解释最适合使用它们的应用程序类型。我们比较了格式化数据文件处理和原始数据文件处理，并介绍了各种基于文件的数据存储机制，比如 `shelve` 和 `cPickle` 模块。

14.2 数据层次结构

计算机处理的所有数据项最终都可简化为 0 和 1 的组合，这是因为可以简单、经济地制造出具有稳定双态的电子设备——0 代表一种状态，1 代表另一种。计算机的所有功能都只牵涉到最基本的 0/1 操作。

计算机支持的最小数据项是“位”（即 bit 或“比特”），每个位只有 0 或 1 这两个值。计算机电路执行各种位操作，比如检查位值、设置位值以及反转位值（0 和 1 互换）。

用最低级的位形式进行编程无疑非常繁琐。更适宜的数据形式包括十进制数位（0, 1, 2, 3, 4, 5, 6, 7, 8 和 9）、字母（A~Z, a~z）以及特殊符号（比如 \$, @, %, &, *, (,), -, +, ", :, ? 和 / 等等）。数位、字母和特殊符号统称“字符”。在特定计算机上，用于编程和表示数据项的所有字符的集合称为该计算机的“字符集”。由于计算机只能处理 0 和 1，所以字符集中的每个字符都用 0 和 1 的组合来表示。“字节”由 8 位构成。程序员用字符创建程序和数据项；计算机则以位形式来操纵和处理这些字符。

字符是由位构成的，同理，“字段”由字符（或字节）构成。字段是一组表达特定含义的字符。例如，只包括大写和小写字母的字段可用于表示人名。由计算机来处理的不同数据项构成了“数据层次结构”

（如图 14.1 所示）。在这个结构中，数据项变得越来越大，越来越复杂——由位变成字符，变成字段，再变成越来越大的数据结构。通常，一条“记录”（比如 Visual Basic 中的一个类）由几个字段（在 Visual Basic 中称为成员变量）构成。比如在工资发放系统中，特定员工的记录可能包括以下字段：

1. 员工标识号
2. 姓名
3. 地址
4. 每小时工资
5. 免税金额
6. 年收入

7. 代扣税

所以，记录是相关字段的一个组合。上例中每个字段都代表员工的一项信息。公司为每名员工都保留一条工资发放记录。“文件”则是相关记录的组合。^①在公司的工资发放文件中，通常为每名员工都包含了一条记录。所以，小公司的工资发放可能只包含 22 条记录，而大公司的同一个文件可能包含 100 000 条记录。一个公司拥有大量文件的情况也并不罕见，有的文件甚至包含数以百万、十亿、万亿计的字符。

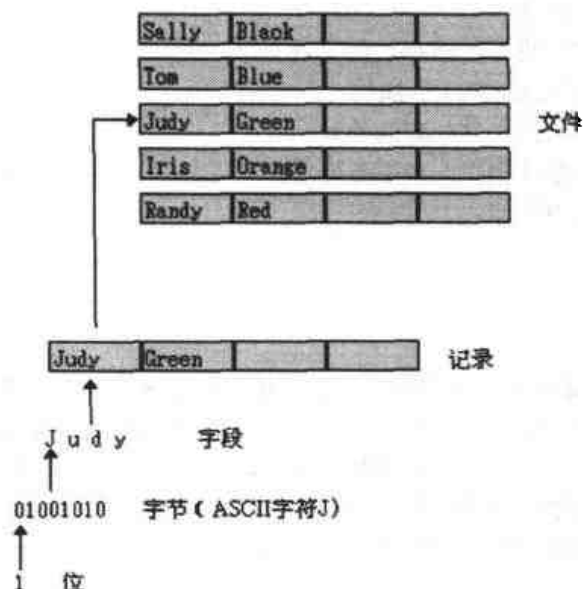


图 14.1 数据层次结构

要从文件中提取特定记录，每条记录至少要有有一个字段作为“记录键”。记录键标识了属于特定人员或实体的记录，并将该记录和文件中的其他所有记录区分开。以前面的工资发放记录为例，通常应选择员工标识号作为记录键。

可通过多种方式对文件中的记录进行组织。最常见的组织形式是“顺序文件”。其中，各记录通常按“记录键”字段顺序存储。在工资发放文件中，通常按员工标识号的顺序放置记录。文件的第一条员工记录包含了最小员工标识号，后续记录包含的员工标识号依次增大。

一般要用不同文件来存储数据。例如，一家公司可使用工资发放文件、应收款文件（列出要向客户收取的款项）、应付款文件（列出应付给供应商的款项）、库存文件（列出所有货物的信息）以及其他多种类型的文件。一组相关文件有时统称为“数据库”。用于创建和管理数据库的一系列程序则统称为“数据库管理系统”（DBMS）。第 17 章将详细讨论数据库。

14.3 文件和流

Python 将每个文件都视为一个顺序的字节流（参见图 14.2）。每个文件都结束于一个 EOF（文件尾）标记，或结束于一个特定字节编号（由系统维护的一个管理数据结构记录）。程序“打开”文件时，Python 会创建一个对象，并将一个“流”与那个对象关联。



图 14.2 Python 所见的一个包含 n 字节的文件

^① 文件通常可以包含任何格式的任何数据。有的操作系统将文件视为字节的集合。在这种操作系统中，文件中的任何字节组织形式（比如将数据组织成记录）都是由程序员创建的一个“视图”。

Python 程序开始执行时，会创建 3 个文件流，包括 `sys.stdin`（标准输入流）、`sys.stdout`（标准输出流）以及 `sys.stderr`（标准错误流）。这些流在程序和特定文件/设备之间建立了沟通渠道。无论 Python 程序是否导入 `sys` 模块，都会创建 Python 文件流（虽然程序必须导入 `sys` 模块才能直接访问流）。程序输入对应于 `sys.stdin`。事实上，`raw_input` 正是借助 `sys.stdin` 来获取用户输入。程序输出对应于 `sys.stdout`。`print` 语句默认将信息发送给标准输出流。程序错误则打印到 `sys.stderr`。

利用 `sys.stdin` 流，程序可从键盘或其他设备接收输入。`sys.stdout` 允许程序将数据输出至屏幕或其他设备。而 `sys.stderr` 流允许程序将错误消息输出至屏幕或其他设备。

14.4 创建顺序访问文件

Python 对文件结构没有要求。像“记录”这样的概念在 Python 文件中是不存在的。换言之，要想满足应用程序的要求，程序员必须自行定义文件结构。在本节的例子中，我们将为文件设置一个记录结构。

图 14.3 创建了一个简单的顺序访问文件，“应收款”系统可用它跟踪公司客户欠款。针对每个客户，程序都获得一个账户编号、客户姓名以及账户余额（即客户欠款金额）。为每个客户获得的数据构成了与该客户对应的一条记录。在这个应用程序中，账户编号代表记录键；换言之，文件将按照账户编号的顺序创建和维护。在我们的例子中，假定用户按账户编号的顺序输入账户信息。在一个完善的应收款系统中，排序功能是必不可少的，它允许用户按任意顺序输入记录，然后再在对记录进行排序之后，将其写入文件。

```
1 # Fig. 14.3: fig14_03.py
2 # Opening and writing to a file.
3
4 import sys
5
6 # open file
7 try:
8     file = open("clients.dat", "w") # open file in write mode
9 except IOError, message:          # file open failed
10     print >> sys.stderr, "File could not be opened:", message
11     sys.exit(1)
12
13 print "Enter the account, name and balance."
14 print "Enter end-of-file to end input."
15
16 while 1:
17     try:
18         accountLine = raw_input( "? " ) # get account entry
19     except EOFError:
20         break # user entered EOF
21     else:
22         print >> file, accountLine # write entry to file
23
24
25 file.close()
```

```
Enter the account, name and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

图 14.3 用于打开文件和写入数据的文件流对象

如前所述，程序员要创建文件流对象以打开文件。`open` 函数取得一个必需参数，以及两个可选参数，

从而创建一个流对象（第 8 行）。对于新的流对象，必需的参数是“文件名”；两个可选参数是“文件打开模式”和“缓冲模式”。

文件打开模式指出文件打开后要如何进行读取，写入，还是两者兼而有之。如果设为“w”（写入），会打开一个文件，以便将数据输出到文件中。以“w”模式打开现有文件，文件中的所有数据都会被删除，然后用新数据重新创建文件。如果指定的文件不存在，则新建一个文件。新文件的名称由“文件名”参数指定（即 clients.dat）。如果没有在文件名参数中指定文件位置，Python 会在当前目录创建文件。如果没有指定“文件打开模式”参数，就默认为“r”，即打开文件以便读取。图 14.4 总结了各种文件打开模式。open 函数的第 3 个参数指定缓冲模式，用于对文件的输入和输出进行高级控制，而且通常都不必指定。本例没有特别指定缓冲模式参数。

模式	说明
"a"	所有输出都写到文件尾。如果指定的文件不存在，就创建一个
"r"	打开文件以便输入。如果文件不存在，就引发 IOError 异常
"r+"	打开文件以便输入和输出。如果文件不存在，就引发 IOError 异常
"w"	打开文件以便输出。如果文件存在，就删除其中所有数据。如果文件不存在，就创建一个
"w+"	打开文件以便输入和输出。如果文件存在，就删除其中所有数据。如果文件不存在，就创建一个
"ab", "rb", "r+b", "wb", "w+b"	打开文件以便进行二进制（也就是非文本形式）输入或输出。注意，只有 Windows 和 Macintosh 平台才支持这些模式

图 14.4 文件打开模式

常见编程错误 14.1 在用户希望保留原始文件内容的前提下，用“w”模式打开文件以进行输出，是一个逻辑错误，因为文件内容会在用户未得到警告的情况下删除。

open 函数遇到错误，会引发 IOError 异常。可能的错误包括试图打开一个不存在的文件进行读取，打开一个只读文件进行写入，以及在磁盘空间不足的时候打开一个文件进行写入。

在图 14.3 中，如果 open 引发 IOError 异常，第 10 行就将错误消息“File could not be opened”打印到 sys.stderr。默认情况下，print 语句将输出发送到 sys.stdout 文件对象。程序可将来自 print 语句的输出“重定向”至一个不同的文件对象。在我们的例子中，以下语句：

```
print >> sys.stderr, "File could not be opened:", message
```

会将输出重定向至 sys.stderr（标准错误）文件对象。如果在 print 关键字之后插入 >> 符号，print 语句会将输出重定向至 >> 右侧指定的文件对象。输出文件对象之后是一个逗号和要打印的值。

常见编程错误 14.2 用 >> 重定向文件输出时，如果忘记在文件对象后插入逗号，会造成语法错误。

用 >> 重定向输出的功能是在 Python 2.0 中加入的。在老版本中，或在希望支持多个版本的程序中，同样的功能也可用文件对象的 write 方法实现，如下所示：

```
sys.stderr.write( output )
```

图 14.5 的表格总结了文件对象的各种方法。图 14.3 的程序在打开文件时如果出错，sys.exit 函数（第 11 行）会终止程序。sys.exit 把它的可选参数返回给程序调用时的环境。参数 0（默认值）表明程序正常终止；其他任何值都表明程序是因为一个错误而终止的。调用环境（极有可能是操作系统）利用 sys.exit 的返回值对错误进行恰当的响应。

如果 clients.dat 文件成功打开，程序会处理数据。第 13~14 行提示用户为每条记录输入相应的字段值；数据输入完毕后，可输入 EOF 标记结束。

方法	说明
<code>close()</code>	关闭文件对象
<code>fileno()</code>	以一个整数的形式返回文件说明符（即操作系统用于维护文件信息的数字）
<code>flush()</code>	刷新文件的缓冲区。缓冲区包含等待写入或从文件中读取的信息。“刷新”就是执行实际的读取或写入操作
<code>isatty()</code>	如果文件对象是 <i>tty</i> （终端）设备，就返回 1
<code>read([size])</code>	从文件中读取数据。如果没有指定 <i>size</i> ，方法读取直到文件尾的所有数据。如果指定了 <i>size</i> ，最多从文件中读取指定的字节数
<code>readline([size])</code>	从文件中读取一行。如果没有指定 <i>size</i> ，方法读取到行尾的所有数据。如果指定了 <i>size</i> ，最多只读取指定的字节数
<code>readlines([size])</code>	从文件中读取多行，并以列表形式返回这些行。如果没有指定 <i>size</i> ，方法读取直到文件尾的所有数据。如果指定了 <i>size</i> ，就最多从文件读取指定的字节数
<code>seek(offset[, location])</code>	使文件位置移动 <i>offset</i> 个字节。如果没有指定 <i>location</i> ，文件位置从文件起始处移动。如果指定了 <i>location</i> ，就从指定位置移动。14.5 节将详细讲解 <i>seek</i>
<code>tell()</code>	返回文件的当前位置
<code>truncate([size])</code>	删除文件中的数据。如果没有指定 <i>size</i> ，就删除所有数据；如果指定了 <i>size</i> ，就最多只删除指定的字节数
<code>write(output)</code>	将字符串 <i>output</i> 写入文件
<code>writelines(outputList)</code>	将 <i>outputList</i> 中的每个字符串写入文件
<code>xreadlines()</code>	类似于 <i>readlines</i> ，但该方法能更有效地利用内存来读取文件。它将返回一个 <i>xreadlines</i> 对象，程序可遍历它以获取信息。 <i>xreadlines</i> 对象详情见于 www.python.org/doc/current/lib/module-xreadlines.html

图 14.5 文件对象的方法

第 16~23 行在重复结构中使用一个 *try/except/else* 块，从标准输入提取每个数据集。*raw_input* 函数从用户处获取一行输入。如果用户输入 EOF 字符，*raw_input* 会引发 *EOFError* 异常。第 20~21 行捕捉这个错误，并用 *break* 语句退出无穷的 *while* 循环。如果用户没有输入 EOF 字符，会执行 *else* 块（第 22~23 行），并使用 *>>* 符号，将用户输入的行打印到输出文件。

close 方法（第 25 行）在 *while* 循环终止后关闭文件。尽管 Python 会在程序终止时关闭打开的文件，但作为一个好习惯，一旦程序不再需要文件，就应该使用 *close* 方法将其关闭。

性能提示 14.1 一旦程序不再需要引用文件，就显式地关闭文件。这样可节省程序所用的资源，还有助于改善程序的可读性。

在图 14.3 的程序的示范执行中，用户输入 5 个账户的信息，并输入 EOF 字符（本例是按 *Ctrl+Z*）以结束输入。这个对话没有显示数据记录在文件中的实际样子。为验证文件已成功创建，下一节将用一个程序来读取文件，并显示它的内容。

14.5 从顺序访问文件读取数据

数据存储到文件后，以后可获取这些数据以进行处理。上一节演示了如何创建顺序访问文件。本节要讨论如何从文件中顺序读取（获取）数据。

图 14.6 从 *clients.dat* 文件中读取记录（该文件是由图 14.3 的程序创建的），并显示每条记录的内容。要想打开文件以进行读取，需要为 *open* 函数的第二个参数传递“*r*”（第 8 行）。如果 *open* 的文件名参数没有指定文件位置，Python 会自动在当前目录查找文件。


```

1 # Fig. 14.6: fig14_06.py
2 # Reading and printing a file.
3
4 import sys
5
6 # open file
7 try:
8     file = open( "clients.dat", "r" )
9 except IOError:
10     print >> sys.stderr, "File could not be opened"
11     sys.exit( 1 )
12
13 records = file.readlines() # retrieve list of lines in file
14
15 print "Account".ljust( 10 ),
16 print "Name".ljust( 10 ),
17 print "Balance".rjust( 10 )
18
19 for record in records: # format each line
20     fields = record.split()
21     print fields[ 0 ].ljust( 10 ),
22     print fields[ 1 ].ljust( 10 ),
23     print fields[ 2 ].rjust( 10 )
24
25 file.close()

```

Account	Name	Balance
100	Jones	24.98
200	Doe	345.67
300	White	0.00
400	Stone	-42.16
500	Rich	224.62

图 14.6 从顺序访问文件中读取数据

文件对象打开后默认就是进行读取，所以以下语句：

```
file = open( "clients.dat" )
```

也能打开 `clients.dat` 进行读取。

良好编程习惯 14.1 如果不希望文件内容被修改，应以只读模式打开文件（使用“r”），防止因为不慎而修改文件内容。这是“最小权限原则”的一个例子。

`readlines` 方法（第 13 行）将整个文件的内容都读入程序。该方法返回一个由文件的各行构成的列表。然后，将该列表存储到变量 `records` 中。针对文件中的每一行（记录），`split` 方法都将行中的单词（字段）作为一个列表返回。第 19~23 行输出这些字段。`ljust` 和 `rjust` 方法分别对字段进行左对齐和右对齐，从而格式化输出。`close` 方法（第 25 行）则关闭与文件对象对应的文件。

Python 2.2 新增了一些特性，允许程序员在 `for` 语句中使用文件对象。例如在一个使用 Python 2.2 的程序中，上例的第 19 行可替换为：

```
for record in file:
```

这样可每次读取 `file` 的一行，并将该行指派给 `record`。程序可立即处理那一行。与通过 `readlines` 方法读取一个大文件的内容相比，用这种方式遍历文件中的各行显得更高效。因为如果使用 `readlines`，必须先将整个文件都读入内存，然后才能对文件内容进行处理。

为了从文件中顺序地获取数据，程序通常要从文件起始处（文件头）开始，连续读取所有数据，直至发现所需的数据。程序执行期间，有时需要多次顺序处理一个文件（都从文件头开始）。为解决这个问题，文件对象提供了 `seek` 方法，可用它重新定位“文件位置指针”（其中包含要从文件中读写的下一个字节的“字节编号”）。例如以下语句：


```
file.seek( 0, 0)
```

会将“文件位置指针”重新定位至文件头。seek 的第一个参数是“偏移量”，它是一个整数值，以字节数的形式指定了文件中的一个位置（要依据文件的“seek 方向”）。第二个参数是可选的，指定偏移的开始“位置”，或称“seek 方向”。seek 方向可设为 0（默认值），表明相对于文件头进行定位；也可设为 1，相对于当前位置进行定位；或设为 2，表明相对于文件尾进行定位。下面提供了定位文件位置指针的一些例子：

```
# position to the nth byte of file
# assumes seek direction is 0
file.seek( n )

# position n bytes forward in file
file.seek( n, 1 )

# position n bytes backward from end of file
file.seek( -n, 2 )

# position at end of file
file.seek( 0, 2 )
```

文件对象的 tell 方法返回文件位置指针的当前位置。以下语句将当前文件位置指针值指派给变量 location：

```
location = file.tell()
```

图 14.7 在程序中使用了 seek，它允许信用部经理显示具有零余额（不欠一分钱）、贷方余额（公司欠客户钱）以及借方余额（客户欠公司钱）的客户的账户信息。程序显示一个菜单，允许经理输入 3 个选项之一，以获得需要的信用信息。选项 1 生成具有零余额的账户列表（第 56~57 行），选项 2 生成具有贷方余额的账户列表（第 58~59 行），选项 3 生成具有借方余额的账户列表（第 60~61 行），选项 4 则终止程序执行（第 62~63 行）。如果输入无效选项，程序会提示用户输入正确的选项（第 64~65 行）。

```
1 # Fig. 14.7: fig14_07.py
2 # Credit inquiry program.
3
4 import sys
5
6 # retrieve one user command
7 def getRequest():
8
9     while 1:
10         request = int( raw_input( "\n? " ) )
11
12         if 1 <= request <= 4:
13             break
14
15     return request
16
17 # determine if balance should be displayed, based on type
18 def shouldDisplay( accountType, balance ):
19
20     if accountType == 2 and balance < 0:    # credit balance
21         return 1
22
23     elif accountType == 3 and balance > 0:  # debit balance
24         return 1
25
26     elif accountType == 1 and balance == 0: # zero balance
27         return 1
28
29     else: return 0
30
31 # print formatted balance data
32 def outputLine( account, name, balance ):
```

```

33
34     print account.ljust( 10 ),
35     print name.ljust( 10 ),
36     print balance.rjust( 10 )
37
38 # open file
39 try:
40     file = open( "clients.dat", "r" )
41 except IOError:
42     print >> sys.stderr, "File could not be opened"
43     sys.exit( 1 )
44
45 print "Enter request"
46 print "1 - List accounts with zero balances"
47 print "2 - List accounts with credit balances"
48 print "3 - List accounts with debit balances"
49 print "4 - End of run"
50
51 # process user request(s)
52 while 1:
53
54     request = getRequest() # get user request
55
56     if request == 1:      # zero balances
57         print "\nAccounts with zero balances:"
58     elif request == 2:    # credit balances
59         print "\nAccounts with credit balances:"
60     elif request == 3:    # debit balances
61         print "\nAccounts with debit balances:"
62     elif request == 4:    # exit loop
63         break
64     else: # getRequest should never let program reach here
65         print "\nInvalid request."
66
67     currentRecord = file.readline() # get first record
68
69     # process each line
70     while ( currentRecord != "" ):
71         account, name, balance = currentRecord.split()
72         balance = float( balance )
73
74         if shouldDisplay( request, balance ):
75             outputLine( account, name, str( balance ) )
76
77         currentRecord = file.readline() # get next record
78
79     file.seek( 0, 0 ) # move to beginning of file
80
81 print "\nEnd of run."
82 file.close() # close file

```

```

Enter request
1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - End of run

```

```
? 1
```

```
Accounts with zero balances:
300      White      0.0
```

```
? 2
```

```
Accounts with credit balances:
400      Stone     -42.16
```

```
? 3
```

```
Accounts with debit balances:
100      Jones     24.98
```

```

200      Doe      345.67
500      Rich      224.62

? 4

End of run.
```

图 14.7 信用查询程序

第 52~77 行对选项 4 之外的所有请求进行处理。`readline` 方法（第 67 行）从文件读取一行，并将文件位置指针移至下一行。`readline` 方法从文件读取所有行之（即程序抵达文件尾），`readline` 会返回空字符串（`""`）。

`split` 方法（第 71 行）将每条记录分解成 3 个变量，即 `account`、`name` 和 `balance`。如果要显示记录，程序就会调用函数 `shouldDisplay`（第 18~29 行），该函数会返回 1（`true`）。如有必要，函数 `outputLine`（第 32~36 行）会显示记录中的各个字段。

14.6 更新顺序访问文件

修改格式化和写入顺序访问文件的数据（如以前的小节所示）时，可能会破坏文件中的其他数据。例如，假定要将“White”修改成“Williams”，就不能简单地覆盖原来的名字。在图 14.7 中，White 的那条记录写入文件后将如下所示：

```
300 White 0.00
```

如果使用“Williams”覆盖该记录，就会变成：

```
200 Williams00
```

新名字包含的字符数比原来的多 3 个，所以在“Williams”中，第 2 个“i”之后的字符会覆盖行内的其他字符。这里的问题在于，在格式化的输入/输出模型中，字段（记录）的长度可能变化。例如，7，14，-117，2 074 和 27 383 都是整数，而且内部都用相同的“原始数据”字节数存储，但这些整数作为格式化的文本（字符序列）输出到屏幕或文件时，会成为不同长度的字段。因此，通常不能用格式化输入/输出模型在文件中当场更新记录。

虽然能在顺序访问文件中更新数据，但显得很笨拙。例如，要进行前述的更名操作，300 White 0.00 之前的记录可拷贝到一个单独的文件。然后，更新的记录可以写入这个文件，再将 300 White 0.0 之后的记录拷贝到这个文件。但整个过程显得非常繁琐，因为更新一条记录就需要对文件中的所有记录进行处理。如果要同时更新文件中的许多记录，使用这种技术将让人无法接受。

14.7 随机访问文件

前面解释了如何创建顺序访问文件，以及如何搜索这些文件以定位特定信息。但对于所谓“即时访问”应用程序，顺序访问文件是不合适的。在这种应用程序中，必须立即定位特定记录。流行的即时访问应用程序包括航空公司订票系统、银行业务系统、电子收款（POS）系统、自动柜员机（ATM）以及其他需要快速访问特定数据的“事务处理系统”。您的开户银行可能有几十乃至几百万个客户；但是，当您使用一台 ATM 时，瞬间即可检查出您的账户上还有多少余额。即时访问可通过“随机访问文件”（有时称为“直接访问文件”）来实现。在这种文件中，可直接和快速地访问单独记录，无需先搜索其他记录。

正如本章前面讨论的那样，Python 不会为文件强加任何结构限制。所以，使用随机访问文件的应用程序必须自行创建随机访问机制。有多种技术可用于创建随机访问文件。最简单的技术要求文件中的所有记录都具有固定长度。使用长度固定的记录，程序可计算任何记录相对于文件头的确切位置（作为记

录长度和记录键的一项功能)。稍后要演示如何利用这种技术快速访问特定记录(即使在大文件中)。

图 14.8 展示了一个由固定长度的记录构成的随机访问文件的视图。每个记录长 100 字节。随机访问文件就像一列火车,它有许多车厢——有的为空,有的满载货物。数据可直接插入随机访问文件,不会破坏文件中的其他数据。此外,以前存储的数据可更新或删除,不用重写整个文件。在后面的小节,将解释如何创建一个随机访问文件,将数据输入那个文件、读取数据、更新数据以及删除不需要的数据。

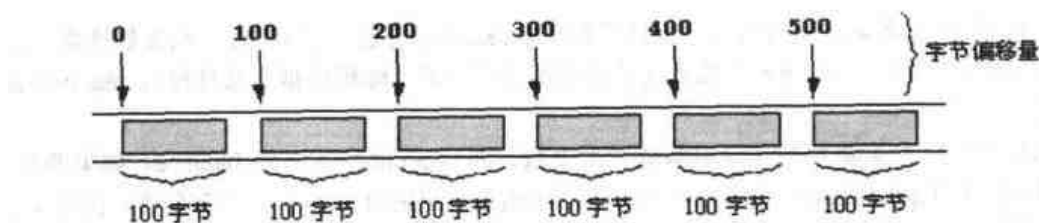


图 14.8 随机访问文件的结果

14.8 模拟随机访问文件: shelve 模块

处理随机访问文件的程序很少将单个字段写入文件。它们往往采取每次写一个记录(或对象)的方式。可用其他程序语言定义一个类,用它表示要写入文件的记录,从而创建随机访问文件。在这种程序语言中,程序要根据类的长度在文件中读写类的实例(类的长度是指类的一个实例所占用的字节数)。Python 提供了 shelve 模块模拟这种行为,利用它,程序员就不必另外写入一个新类。后续的例子将使用这个模块。

下面是一个信用处理应用程序的问题陈述:

为最多有 100 名客户的一家公司创建一个事务处理程序,它最多能存储 100 个定长记录。每个记录都包括账号(作为“记录键”使用)、姓氏、名字以及余额。程序可更新账户、插入新账户、删除账户以及列出文件中的所有账户记录。

后面几节要介绍创建这个程序所需的技术。可用 shelve 模块在文件中读写记录。为此,要创建 shelve 对象来表示记录。这些对象具有一个字典接口——也就是访问记录信息的记录键。在我们的例子中,记录键就是账户号码,而记录值是一个列表,其中包含客户姓氏、名字和账户余额。

14.9 将数据写入 shelve 文件

图 14.9 从用户处获取账户信息,并将数据写入 shelve 文件 credit.dat 中。第 9 行使用 shelve.open 函数打开 credit.dat 文件。该函数类似于 Python 函数 open(该函数用于打开普通文本)。如果文件不存在,函数会新建文件。如果打开文件时出错,函数引发 IOError 异常。

```
1 # Fig. 14.9: fig14_09.py
2 # Writing to shelve file.
3
4 import sys
5 import shelve
6
7 # open shelve file
8 try:
9     outCredit = shelve.open("credit.dat")
10 except IOError:
11     print ">> sys.stderr, \"File could not be opened\"
12     sys.exit(1)
13
14 print "Enter account number (1 to 100, 0 to end input)"
```

```

15
16 # get account information
17 while 1:
18
19     # get account information
20     accountNumber = int( raw_input(
21         "\nEnter account number\n? " ) )
22
23     if 0 < accountNumber <= 100:
24
25         print "Enter lastname, firstname, balance"
26         currentData = raw_input( "? " )
27
28         outCredit[ str( accountNumber ) ] = currentData.split()
29
30     elif accountNumber == 0:
31         break
32
33 outCredit.close() # close shelve file

```

```

Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00

Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54

Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98

Enter account number
? 88

Enter lastname, firstname, balance
? Smith Dave 258.34

Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33

Enter account number
? 0

```

图 14.9 写入 shelve 文件的数据

第 20~21 行提示用户输入 1~100 的账户号码。第 28 行将数据写入 shelve 文件。程序通过字典接口处理 shelve 文件中的数据。shelve 文件中的每个键都必须是一个字符串，所以，str 函数将整数值 accountNumber 转换成字符串（第 28 行）。split 方法将用户输入的数据转换成一个列表，这些数据保存为记录键的值（第 28 行）。输入 0 表明数据结束，文件对象 close 方法会关闭 shelve 文件（第 33 行）。

14.10 从 shelve 文件获取数据

上节创建一个 shelve 文件并将数据写入其中。本节的例子（图 14.10）要遍历文件并打印其中的每个记录。

```

1 # Fig. 14.10: fig14_10.py
2 # Reading shelve file.
3
4 import sys

```

```

5 import shelve
6
7 # print formatted credit data
8 def outputLine( account, aList ):
9
10     print account.ljust( 10 ),
11     print aList[ 0 ].ljust( 10 ),
12     print aList[ 1 ].ljust( 10 ),
13     print aList[ 2 ].rjust( 10 )
14
15 # open shelve file
16 try:
17     creditFile = shelve.open( "credit.dat" )
18 except IOError:
19     print >> sys.stderr, "File could not be opened"
20     sys.exit( 1 )
21
22 print "Account".ljust( 10 ),
23 print "Last Name".ljust( 10 ),
24 print "First Name".ljust( 10 ),
25 print "Balance".rjust( 10 )
26
27 # display each account
28 for accountNumber in creditFile.keys():
29     outputLine( accountNumber, creditFile[ accountNumber ] )
30
31 creditFile.close() # close shelve file

```

Account	Last Name	First Name	Balance
37	Barker	Doug	0.00
88	Smith	Dave	258.34
33	Dunn	Stacey	314.33
29	Brown	Nancy	-24.54
96	Stone	Sam	34.98

图 14.10 从 shelve 文件读取数据

`keys` 方法返回 shelve 文件中的记录键的一个列表（第 28 行）。for 循环遍历列表，将每个记录键及其值传给 `outputLine` 函数。该函数（第 8~13 行）打印记录键及其相应的值。

14.11 示例：一个事务处理程序

现在来开发一个实际的事务处理程序（图 14.11），它用一个 shelve 文件实现“即时访问”。程序维护银行的账户信息。用户可更新现有账户、添加新账户、删除账户，并可当前账户的格式化列表存储到文本文件中。

```

1 # Fig. 14.11: fig14_11.py
2 # Reads shelve file, updates data
3 # already written to file, creates data
4 # to be placed in file and deletes data
5 # already in file.
6
7 import sys
8 import shelve
9
10 # prompt for input menu choice
11 def enterChoice():
12
13     print "\nEnter your choice"
14     print "1 - store a formatted text file of accounts"
15     print "   called \"print.txt\" for printing"
16     print "2 - update an account"
17     print "3 - add a new account"
18     print "4 - delete an account"
19     print "5 - end program"

```

```

20
21 while 1:
22     menuChoice = int( raw_input( "? " ) )
23
24     if not 1 <= menuChoice <= 5:
25         print >> sys.stderr, "Incorrect choice"
26
27     else:
28         break
29
30     return menuChoice
31
32 # create formatted text file for printing
33 def textFile( readFromFile ):
34
35     # open text file
36     try:
37         outputFile = open( "print.txt", "w" )
38     except IOError:
39         print >> sys.stderr, "File could not be opened."
40         sys.exit( 1 )
41
42     print >> outputFile, "Account".ljust( 10 ),
43     print >> outputFile, "Last Name".ljust( 10 ),
44     print >> outputFile, "First Name".ljust( 10 ),
45     print >> outputFile, "Balance".rjust( 10 )
46
47     # print shelf values to text file
48     for key in readFromFile.keys():
49         print >> outputFile, key.ljust( 10 ),
50         print >> outputFile, readFromFile[ key ][ 0 ].ljust( 10 ),
51         print >> outputFile, readFromFile[ key ][ 1 ].ljust( 10 ),
52         print >> outputFile, readFromFile[ key ][ 2 ].ljust( 10 )
53
54     outputFile.close()
55
56 # update account balance
57 def updateRecord( updateFile ):
58
59     account = getAccount( "Enter account to update" )
60
61     if updateFile.has_key( account ):
62         outputLine( account, updateFile[ account ] ) # get record
63
64         transaction = raw_input(
65             "\nEnter charge (+) or payment (-): " )
66
67         # create temporary record to alter data
68         tempRecord = updateFile[ account ]
69         tempBalance = float( tempRecord[ 2 ] )
70         tempBalance += float( transaction )
71         tempBalance = "%.2f" % tempBalance
72         tempRecord[ 2 ] = tempBalance
73
74         # update record in shelf
75         del updateFile[ account ] # remove old record first
76         updateFile[ account ] = tempRecord
77         outputLine( account, updateFile[ account ] )
78     else:
79         print >> sys.stderr, "Account #", account, \
80             "does not exist."
81
82 # create and insert new record
83 def newRecord( insertInFile ):
84
85     account = getAccount( "Enter new account number" )
86
87     if not insertInFile.has_key( account ):
88         print "Enter lastname, firstname, balance"
89         currentData = raw_input( "? " )
90         insertInFile[ account ] = currentData.split()

```

```

91     else:
92         print >> sys.stderr, "Account #", account, "exists."
93
94 # delete existing record
95 def deleteRecord( deleteFromFile ):
96
97     account = getAccount( "Enter account to delete" )
98
99     if deleteFromFile.has_key( account ):
100         del deleteFromFile[ account ]
101         print "Account #", account, "deleted."
102     else:
103         print >> sys.stderr, "Account #", account, \
104             "does not exist."
105
106
107 # output line of client information
108 def outputLine( account, record ):
109
110     print account.ljust( 10 ),
111     print record[ 0 ].ljust( 10 ),
112     print record[ 1 ].ljust( 10 ),
113     print record[ 2 ].rjust( 10 )
114
115 # get account number from keyboard
116 def getAccount( prompt ):
117
118     while 1:
119         account = raw_input( prompt + " (1 - 100): " )
120
121         if 1 <= int( account ) <= 100:
122             break
123
124     return account
125
126 # list of functions that correspond to user options
127 options = [ textFile, updateRecord, newRecord, deleteRecord ]
128
129 # open shelve file
130 try:
131     creditFile = shelve.open( "credit.dat" )
132 except IOError:
133     print >> sys.stderr, "File could not be opened."
134     sys.exit( 1 )
135
136 # process user commands
137 while 1:
138
139     choice = enterChoice()           # get user menu choice
140
141     if choice == 5:
142         break
143
144     options[ choice - 1 ]( creditFile ) # invoke option function
145
146 creditFile.close()                 # close shelve file

```

图 14.11 银行账户程序

首先执行图 14.9 的程序，以便在本事务处理程序用到的文件中插入实际数据。事务处理程序为用户提供 5 个选项（1~5）。其中，选项 1 调用函数 `textFile`（第 33~54 行），它将所有账户信息的一个格式化列表存储到文本文件 `print.txt` 中。用户可利用该文件打印账户信息的一个列表。`textFile` 函数取得一个 `shelve` 文件作为参数，并根据那个文件中的数据创建文本文件。`outputLine` 函数（第 108~113 行）将数据输出到文件 `stdout` 中。选择选项 1 后，`print.txt` 将包含以下文本：

Account	Last Name	First Name	Balance
37	Barker	Doug	0.00
88	Smith	Dave	258.54
33	Dunn	Stacey	314.33
29	Brown	Nancy	-24.54
96	Stone	Sam	34.98

如果选择选项 2, 程序就会调用 `updateRecord` 函数 (第 57~80 行) 以更新账户。首先, 函数判断用户指定的记录是否存在 (因为函数只能更新现有记录)。如果记录存在, 就把它读入变量 `tempRecord`。第 69~70 行将账户余额的字符串表示转换成一个浮点值, 以进行数值处理。更新 `shelve` 文件中的记录之前, 程序首先必须删除指定账户的现有记录; 关键字 `del` (第 75 行) 用于删除当前记录。第 76 行更新记录——将新记录值指派给相应的账户号码 (记录键)。然后, 程序输出更新过的值。下面是该选项的一个典型输出:

```
Enter account to update (1 - 100): 37
37      Barker      Doug      0.00

Enter charge (+) or payment (-): +87.99
37      Barker      Doug      87.99
```

移植性提示 14.1 并非所有 Python 平台都要求在更新记录之前, 先从 `shelve` 文件删除该记录的值。然而, 在更新前坚持用 `del` 删除记录值, 可保证在所有 Python 平台上都能正确更新记录。

选项 3 调用函数 `newRecord`, 让用户添加一个新账户。该函数采用与图 14.9 所示的程序相同的方式添加账户。如果用户输入的是一个现有账户的账户号码, `newRecord` 就显示一条消息, 指明该账户存在, 并允许用户选择要执行的下一个操作。下面是选项 3 的一个典型输出:

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

选项 4 调用 `deleteRecord` 函数, 删除不再需要的记录。程序提示用户输入账户号码。如果账户存在, 程序就用关键字 `del` 从 `shelve` 文件删除那个记录, 然后显示一条消息, 指出记录已成功删除。然而, 如果指定的账户不存在, 程序将显示一条错误消息。选项 4 的典型输出如下:

```
Enter account to delete (1 - 100): 29
Account # 29 deleted.
```

选项 5 终止程序。程序的主要部分 (第 127~146 行) 创建一个与用户菜单选项对应的功能列表 (第 127 行)。然后, 打开存储银行账户的 `shelve` 文件, 并让用户选择菜单选项。

第 144 行调用一个对应于用户选择的函数。记住, 圆括号是 Python 运算符。如果与函数名 (例如 `textFile`) 联合使用, 这种运算符就会调用函数, 并传递任何指定的参数。`options` 变量保存函数名的一个列表的, 所以像下面这样的语句:

```
options[0] (creditFile)
```

会调用函数 `textFile` (列表中的第一个函数), 并将 `creditFile` 作为参数传递。这样的语句可避免使用冗长的 `if/else` 语句来判断用户菜单选项, 并调用相应的函数。

14.12 对象序列化

“序列化” (Serialization) 是指将用户自定义类等复杂对象类型转换成字节集, 以便存储或通过网络传输。序列化也称为“平坦化” (Flattening) 或者“编组” (Marshalling)。Python 用 `pickle` 和 `cPickle` 模块来执行序列化。本书使用的是 `cPickle` 模块 (用 C 写成的), 而不使用 `pickle` 模块, 因为在执行时,

用编译型语言（比如 C）写的模块要比用解释型语言（比如 Python）写的模块快。图 14.12 演示了如何对一个列表进行序列化，并把它存储到文件中。

性能提示 14.2 cPickle 模块的执行效率高于 pickle 模块，因为 cPickle 是用 C 实现的，并被编译成每种平台上的原生机器语言。

```

1 # Fig. 14.12: fig14_12.py
2 # Opening and writing pickled object to file.
3
4 import sys, cPickle
5
6 # open file
7 try:
8     file = open( "users.dat", "w" )    # open file in write mode
9 except IOError, message:              # file open failed
10     print >> sys.stderr, "File could not be opened:", message
11     sys.exit( 1 )
12
13 print "Enter the user name, name and date of birth."
14 print "Enter end-of-file to end input."
15
16 inputList = []
17
18 while 1:
19     try:
20         accountLine = raw_input( "? " )    # get user entry
21     except EOFError:
22         break                               # user-entered EOF
23     else:
24         inputList.append( accountLine.split() ) # append entry
25
26
27 cPickle.dump( inputList, file ) # write pickled object to file
28
29 file.close()

```

```

Enter the user name, name and date of birth.
Enter end-of-file to end input.
? mike Michael 4/3/60
? joe Joseph 12/5/71
? amy Amelia 7/10/80
? jan Janice 8/18/74
? ^Z

```

图 14.12 将序列化的对象写入文件

第 8 行打开 users.dat，我们要在其中存储着序列化好的对象。第 16 行初始化的 inputList 是一个包含用户输入信息的列表。第 18~25 行提示用户输入信息，并将其追加到 inputList 中。函数 cPickle.dump（第 27 行）将 inputList 序列化到文件。传给 dump 的第一个参数是要序列化的对象，第二个参数是文件对象，指定要由 dump 方法在其中存储序列化对象的一个文件。函数将 inputList 转换成一系列字节，并将字节流写入文件。第 29 行调用文件对象的 close 方法以关闭文件。

要将序列化的对象转换回原始格式，可对数据进行“反序列化”。图 14.13 演示了这一过程。该例使用的是图 14.12 所示程序创建的序列化文件。

程序首先打开文件 users.dat（第 7~11 行）。函数 cPickle.load（第 13 行）对文件中的数据进行反序列化。函数取得包含序列化对象的一个文件对象参数，将以前序列化好的对象转换成 Python 对象，并返回对该对象的一个引用。我们把这个引用指派给变量 records。然后，程序关闭这个不再需要的文件（第 14 行）。程序剩余的部分（第 16~23 行）通过遍历列表来显示反序列化的数据。

```

1 # Fig. 14.13: fig14_13.py
2 # Reading and printing pickled object in a file.
3
4 import sys, cPickle

```

```
5
6 # open file
7 try:
8     file = open( "users.dat", "r" )
9 except IOError:
10     print >> sys.stderr, "File could not be opened"
11     sys.exit( 1 )
12
13 records = cPickle.load( file ) # retrieve list of lines in file
14 file.close()
15
16 print "Username".ljust( 15 ),
17 print "Name".ljust( 10 ),
18 print "Date of birth".rjust( 20 )
19
20 for record in records:      # format each line
21     print record[ 0 ].ljust( 15 ),
22     print record[ 1 ].ljust( 10 ),
23     print record[ 2 ].rjust( 20 )
```

Username	Name	Date of birth
mike	Michael	4/3/60
joe	Joseph	12/5/71
amy	Amelia	7/10/80
jan	Janice	8/18/74

图 14.13 从文件读入序列化的对象

第 15 章 可扩展标记语言 (XML)

学习目标

- 理解 XML
- 会用 XML 标记数据
- 熟悉用 XML 创建的标记语言的类型
- 理解 DTD、Schema 和 XML 之间的关系
- 理解基于 DOM 和基于 SAX 的解析基础
- 理解 XML 命名空间的概念
- 会创建简单的 XSLT 文档

15.1 概述

XML (可扩展标记语言, Extensible Markup Language) 于 1996 年由万维网协会 (W3C) 下属的“XML 工作组”开发成功。XML 是一种可移植的、获得普遍支持的开放式技术 (也就是一种非专利技术), 它用于对数据进行描述。XML 很快就成为在应用程序之间交换数据的一个标准。使用 XML, 文档作者可描述任何类型的数据, 包括数学公式、软件配置指南、乐谱、菜谱以及财务报表。使用 XML 的另一个好处在于, 无论人还是机器, 都可顺利地阅读文档。

本章探讨了 XML 以及各种相关的技术。前 3 节介绍了 XML, 并解释如何用它标记数据。接着两节描述了两个可编程的库, 它们用于对 XML 文档进行处理。在后续的小节, 则介绍了几种 XML “词汇表” (也就是用 XML 创建的标记语言)。本章还简介了一种称为“可扩展样式表语言转换”(XSLT) 的技术, 它将 XML 数据转换成其他基于文本的格式。第 16 章将根据本章介绍的概念, 实际编写使用了 XML 的 Python 应用程序。

15.2 XML 文档

我们的第一个 XML 文档描述了一篇文章 (如图 15.1 所示)。注意, 每个 XML 文档中显示的行号都是为了方便阅读和说明。实际的 XML 文档中并不包含这些行号。

```
1 <?xml version = "1.0"?>
2
3 <!-- Fig. 15.1: article.xml -->
4 <!-- Article structured with XML. -->
5
6 <article>
7
8     <title>Simple XML</title>
9
10    <date>December 21, 2001</date>
11
12    <author>
13        <firstName>John</firstName>
14        <lastName>Doe</lastName>
15    </author>
16
17    <summary>XML is pretty easy.</summary>
18
19    <content>In this chapter, we present a wide variety of examples
20        that use XML.
21    </content>
22
```

```
23 </article>
```

图 15.1 用于标记一篇文章的 XML

这个文档首先是一个可选的“XML 声明”(第 1 行),它将本文档标识成一个 XML 文档。名为 version 的一个“信息参数”指定了本文档中使用的 XML 的版本。^①XML 注释(第 3~4 行)以<!--开头,以-->结尾,而且几乎可放置于 XML 文档的任何地方。和在 Python 程序中一样,XML 注释只用于文档化用途。

常见编程错误 15.1 XML 声明之前出现任何字符(包括空白字符)都是语法错误。

移植性提示 15.1 尽管 XML 声明是可选的,但最好通过声明标识所用的 XML 的版本。否则,未来缺少 XML 声明的文档会假定为遵循最新的 XML 版本,从而导致不可预测的错误。

XML 使用“标记”来标记数据,它们是封闭在一对尖括号(<>)中的名称。标记成对使用,以便对字符数据(比如 Simple XML)进行定界。用于开始一个标记(即 XML 数据)的标记称为“起始标记”,用于终止一个标记的标记则称为“结束标记”。起始标记的例子包括<article>和<title>(分别参见第 6 行和第 8 行)。结束标记与起始标记的区别在于,在<字符之后,紧接着的是一个正斜杠(/)字符。结束标记的例子包括</title>和</article>(分别参见第 8 行和第 23 行)。XML 文档可包含任意数量的标记。

常见编程错误 15.2 忘记为起始标记提供一个相应的结束标记是错误的。

独立的标记单元(即包括在起始标记及结束标记之间的一切内容)称为“元素”。一个 XML 文档总体上只由一个元素构成,即“根元素”,其中包含了文档中的其他所有元素。根元素必须是 XML 声明之后的第一个元素。在图 15.1 中,article(第 6 行)是根元素。不同元素可以嵌套,以形成一个层次结构(根元素位于层次结构的最顶部)。这样一来,文档作者就可在数据之间明确建立关系。例如,title, date, author, summary 和 content 元素都嵌套在 article 内。而 firstName 和 lastName 都嵌套在 author 内。

常见编程错误 15.3 在 XML 文档中创建多个根元素是错误的。

title 元素(第 8 行)包含文章的标题,即 Simple XML,并采用字符数据的形式。类似地,date(第 10 行)、summary(第 17 行)和 content(第 19~21 行)分别包含代表文章发布日期、摘要和内容的字符数据。XML 标记名长度不限,并可采用字符、数位、下划线、连字号以及句点(但必须以字母或下划线开头)。

常见编程错误 15.4 XML 是区分大小写的。记住,不要把 XML 标记名的大小写形式弄错了。

就文档本身来说,它不过是一个名为 article.xml 的文本文件。尽管并非绝对必要,但大多数 XML 文档文件名都以.xml 扩展名结尾。^②为处理 XML 文档,需要一种名为“XML 解析器”的程序。解析器负责检查一个 XML 文档的语法,并使 XML 文档的数据能供应用程序使用。通常,XML 解析器内建于应用程序中,也可通过 Internet 下载。流行的解析器包括微软的 msxml、4DOM(第 16 章要全面使用的一个 Python 包)、Apache Software Foundation 的 Xerces 以及 IBM 的 XML4J。本章使用的是 msxml。

在 Internet Explorer(简称 IE)^③中载入 article.xml 后,msxml 会解析文档,并将解析过的数据传送给 IE。然后,IE 使用一个内建的“样式表”格式化数据。注意,最终的数据格式(图 15.2)与 XML 文档的原始格式(图 15.1)非常相似。正如马上就要展示那样,为了将 XML 数据转换成可显示的格式,样式表担当了至关重要的一个角色。

^① 目前已经升级到 XML 2.0。

^② 用于处理 XML 的一些应用程序要求必须使用这个扩展名。

^③ 要求 IE5 或更高版本。

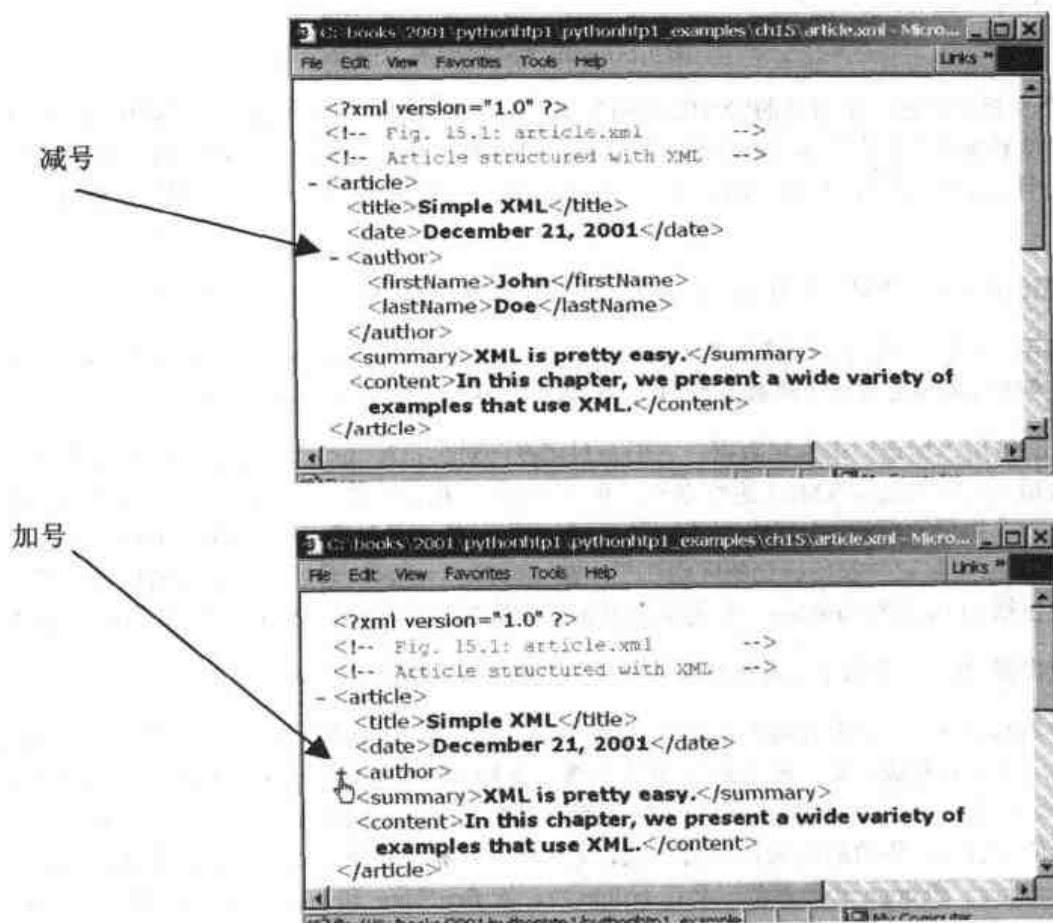


图 15.2 在 Internet Explorer 中显示的 article.xml

注意图 15.2 中出现的减号(-)和加号(+).尽管它们不是 XML 文档的一部分,但 IE 会在所有“容器元素”(包含了其他元素的元素)的旁边显示它们。容器元素也称为“父元素”。减号表明父元素的“子元素”(即嵌套于其中的元素)当前已经显示。减号单击后会变成加号(它会折叠容器元素,隐藏其中的所有子元素)。相反,如果单击加号,会展开容器元素,而且加号会变成减号。这类似于在 Windows 资源管理器中查看目录结构时的操作。事实上,目录结构通常采用一系列树结构的形式,其中每个驱动器字母(比如 C:)都代表一个树的“根”。每个文件夹都是树上的一个“节点”。解析器常常将 XML 数据放到不同的树中,以实现它们的高效处理,详情可参见 15.4 节。

常见编程错误 15.5 不正确地嵌套 XML 标记会导致出错。例如, `<x><y>hello</x></y>` 是错误的,标记 `</y>` 必须位于标记 `</x>` 之前。

下面是第二个示范 XML 文档(参见图 15.3),它标记了一封商务信函。该文档包含的数据远远超过了上一个 XML 文档。

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 15.3: letter.xml -->
4 <!-- Business letter formatted with XML. -->
5
6 <letter>
7     <contact type = "from">
8         <name>Jane Doe</name>
9         <address1>Box 12345</address1>
10        <address2>15 Any Ave.</address2>
11        <city>Othertown</city>

```

```
12     <state>Otherstate</state>
13     <zip>67890</zip>
14     <phone>555-4321</phone>
15     <flag gender = "F" />
16 </contact>
17
18 <contact type = "to">
19     <name>John Doe</name>
20     <address1>123 Main St.</address1>
21     <address2></address2>
22     <city>Anytown</city>
23     <state>Anystate</state>
24     <zip>12345</zip>
25     <phone>555-1234</phone>
26     <flag gender = "M" />
27 </contact>
28
29 <salutation>Dear Sir:</salutation>
30
31     <paragraph>It is our privilege to inform you about our new
32     database managed with <technology>XML</technology>. This
33     new system allows you to reduce the load on
34     your inventory list server by having the client machine
35     perform the work of sorting and filtering the data.
36     </paragraph>
37
38     <paragraph>Please visit our Web site for availability
39     and pricing.
40     </paragraph>
41
42 <closing>Sincerely</closing>
43
44     <signature>Ms. Doe</signature>
45 </letter>
```

图 15.3 标记成 XML 的商务信函

根元素 letter (第 6~45 行) 包含子元素 contact (第 7~16 行和第 18~27 行)、salutation、paragraph、closing 以及 signature。数据除了能放到标记之间, 还可放到“属性”中, 它们是起始标记中的“名称-值”对。每个元素都可在其起始标记中指定任意数量的属性。第一个 contact 元素 (第 7~16 行) 指定了 type 属性, 属性值是“from”, 表明该 contact 元素标记的是与信函发件人有关的信息。第二个 contact 元素 (第 18~27 行) 也指定了 type 属性, 它的值是“to”, 表明这个 contact 元素标记了信函收件人的信息。和标记名一样, 属性名也有几个要求: 要区分大小写、可为任意长度、可采用字母、数位、下划线、连字号和句点、而且必须以字母或下划线字符开头。contact 元素存储了联系人的姓名、地址和电话号码信息。salutation 元素 (第 29 行) 标记的是信函的问候语。第 31~40 行用 paragraph 元素标记了信函正文。closing 元素 (第 42 行) 和 signature 元素 (第 44 行) 则分别标记了信函作者的结语和签名。

常见编程错误 15.6 属性值必须用双引号 (") 或单引号 (') 封闭, 否则就是错误。

第 15 行引入了名为 flag 的一个“空元素”, 它指出联系人的性别。空元素不包含字符数据 (也就是说, 不在起始标记与结束标记之间包含文本)。要想结束这样的元素, 要么在元素末尾插入一个正斜杠 (就像第 15 行那样), 要么显式地写一个结束标记, 比如:

```
<flag gender = "F"></flag>
```

15.3 XML 命名空间

像 Python 这样的语言都提供了各种各样的类库, 并将各种特性分组到不同的命名空间中。这些命

命名空间可防止程序员自定义标识符与类库中的标识符出现“命名冲突”。例如，我们可用 `Book` 类表示与我们的一本出版物有关的信息；但是，集邮爱好者可能用 `Book` 类表示一本集邮册。如果不用命名空间区分这两个 `Book` 类，那么在同一个应用程序中使用它们时，就会发生命名冲突。

类似于 Python，XML 也用“命名空间”对 XML 元素进行区分。此外，对于一些以 XML 为基础的语言（称为“词汇表”）来说，比如 XML Schema（参见 15.6 节）以及可扩展样式表语言（参见 15.8 节），也常常用命名空间标识它们的元素。

命名空间前缀标识元素属于哪个命名空间，从而对元素进行区分。例如：

```
<deitel:publication>
  Python How to Program
</deitel:publication>
```

用命名空间前缀 `deitel` 来限定 `publication` 元素。它表明 `publication` 元素是 `deitel` 这个命名空间的一部分。除了保留的命名空间前缀 `xml` 之外，文档作者还可为命名空间前缀指派任意名称。

常见编程错误 15.7 以任何大小写组合形式创建名为 `xml` 的命名空间前缀都是错误的。

图 15.4 演示了如何使用命名空间。这个 XML 文档包含两个 `file` 元素，它们用命名空间加以区分。

```
1 <?xml version = "1.0"?>
2
3 <!-- Fig. 15.4: namespace.xml -->
4 <!-- Demonstrating namespaces. -->
5
6 <text:directory xmlns:text = "http://www.deitel.com/ns/python1e"
7   xmlns:image = "http://www.deitel.com/images/ns/120101">
8
9   <text:file filename = "book.xml">
10     <text:description>A book list</text:description>
11   </text:file>
12
13   <image:file filename = "funny.jpg">
14     <image:description>A funny picture</image:description>
15     <image:size width = "200" height = "100" />
16   </image:file>
17
18 </text:directory>
```

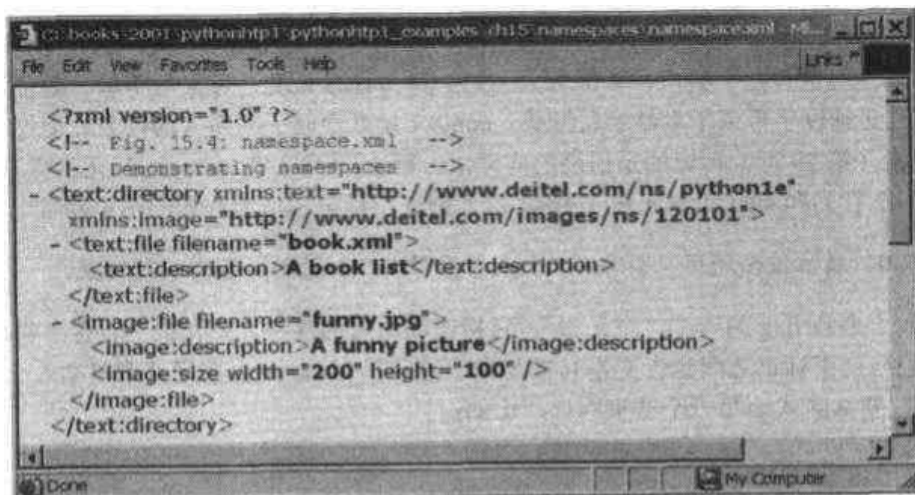


图 15.4 XML 命名空间演示

软件工程知识 15.1 属性无需用命名空间前缀加以限定，因为它们肯定同元素联系在一起。

第 6~7 行使用 `xmlns` 属性创建两个命名空间前缀：`text` 和 `image`。每个命名空间前缀都和一系列名为“统一资源标识符”（URI）的字符绑定。这个 URI 唯一性地标识了命名空间。文档作者可创建自己的

命名空间及 URI。

为确保命名空间是独一无二的，文档作者必须提供惟一的 URI。在此，我们将“<http://www.deitel.com/ns/python1e>”和“<http://www.deitel.com/images/ns/120101>”用作 URI。一种常见的做法是将 URL（统一资源定位符）作为 URI 使用，因为 URL 中使用的域名（比如 www.deitel.com）肯定是没有重复的。这个例子使用与 Deitel & Associates 公司域名对应的 URL 来标识命名空间。注意，解析器永远不会访问这些 URL——它们只是一系列用于区分名称的字符。这里的 URL 并不引用实际的网页，也不一定需要具有正确的形式。

第 9~11 行使用命名空间前缀 text 来描述元素 file 和 description。注意，命名空间前缀 text 也用于结束标记的名称中。第 13~16 行将命名空间前缀 image 应用于元素 file, description 以及 size。

要想避免为每个标记名都附加命名空间前缀，文档作者可考虑指定一个“默认命名空间”。图 15.5 演示了如何创建和使用默认命名空间。

```
1 <?xml version = "1.0"?>
2
3 <!-- Fig. 15.5: defaultnamespace.xml -->
4 <!-- Using default namespaces. -->
5
6 <directory xmlns = "http://www.deitel.com/ns/python1e"
7   xmlns:image = "http://www.deitel.com/images/ns/120101">
8
9   <file filename = "book.xml">
10     <description>A book list</description>
11   </file>
12
13   <image:file filename = "funny.jpg">
14     <image:description>A funny picture</image:description>
15     <image:size width = "200" height = "100" />
16   </image:file>
17
18 </directory>
```

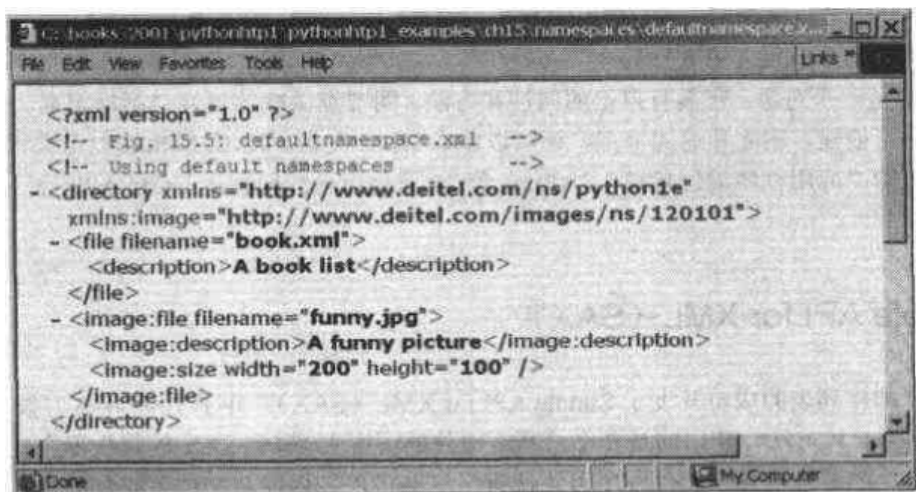


图 15.5 默认命名空间演示

第 6 行将一个 URI 绑定给属性 xmlns，从而定义了一个默认命名空间。一旦定义好默认命名空间，如果子元素中的标记名属于这个命名空间，就不再需要用一个命名空间前缀加以限定。第 9~11 行的 file 元素位于和“<http://www.deitel.com/ns/python1e>”这一 URI 对应的命名空间中。请把它与图 15.4 的第 9~11 行比较，我们在那里为 file 和 description 元素都附加了前缀。

默认命名空间应用于 directory 元素以及所有没有用命名空间前缀限定的元素。但是，也可使用一个命名空间前缀，为特定元素指定一个不同的命名空间。例如，第 13 行为标记名 file 附加了 image 前缀，表明它位于“URI <http://www.deitel.com/images/ns/120101>”所对应的命名空间中，而不是默认命名空间中。

15.4 文档对象模型 (DOM)

尽管 XML 文档本身是文本文件，但通过顺序文件访问技术从中获取数据，显得既不实际，也没有效率，尤其是在其中的数据需要动态添加或删除的时候。

成功解析文档后，有些 XML 解析器会将文档数据以树结构形式存储到内存中。图 15.6 展示了图 15.1 讨论的 `article.xml` 文档的树结构。这个层次化的树结构称为“文档对象模型”(DOM)树，能够创建这类结构的 XML 解析器称为“DOM 解析器”。DOM 树将 XML 文档的每个组件(比如 `article`, `date`, `firstName` 等等)表示成树上的一个节点。包含了其他节点(子节点)的一个节点(比如 `author`)称为父节点。具有同一个父的节点(比如 `firstName` 和 `lastName`)称为“同辈节点”或称“兄弟节点”(Sibling Node)。节点的“后辈节点”(Descendant Node)包括节点的子节点、子节点的子节点……等等。类似地，节点的“前辈节点”(Ancestor Node)包括节点的父节点、父节点的父节点……等等。

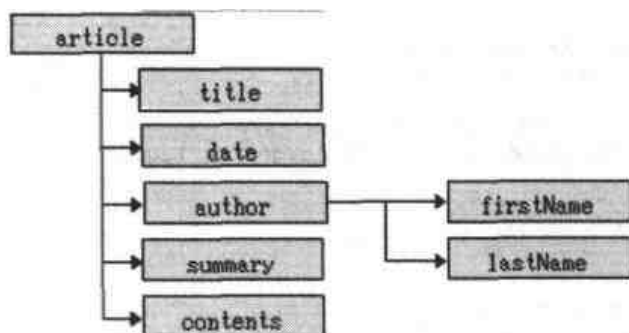


图 15.6 `article.xml` 的树结构

DOM 只有单独一个“根节点”，即“文档根”，其中包含文档中的其他所有节点。例如，`article.xml` (参见图 15.1) 的根节点包含用于 XML 声明的一个节点(第 1 行)，用于注释的两个节点(第 3~4 行)，以及用于根元素的一个节点(第 6 行)。

每个节点都是一个对象，它具有自己的属性和方法。同节点关联的属性包括标记名、值、子节点等。利用方法，程序可创建、删除和追加节点，还可以载入 XML 文档。XML 解析器以可编程的库的形式来提供这些方法，即“应用程序编程接口”(API)。第 16 章将讨论如何使用 DOM API。

15.5 Simple API for XML (SAX)

XML-DEV 邮件列表的成员开发了 Simple API for XML (SAX)，并于 1998 年 5 月发布。SAX 是解析 XML 文档的另一种方法，使用的是一个“基于事件的模型”。基于 SAX 的解析器在处理文档时，会生成名为“事件”的通知信息。软件程序可“侦听”这些事件，以便从文档获取数据。例如，用于生成邮件列表的一个程序可从 XML 文档读取姓名和地址信息，而这个文档包含的并不仅仅是邮寄地址信息，比如还可能包括生日、电话号码和电子邮件地址等等。程序可用 SAX 解析器对文档进行解析，并只侦听包含姓名和地址信息的事件。大量程序语言都可使用基于 SAX 的解析器，比如 Python、Java 和 C++ 等等。我们将在第 16 章演示基于 SAX 的解析。

针对 XML 文档数据的访问，SAX 和 DOM 提供了截然不同的 API。每种 API 都有自己的优点和缺点。DOM 是一种基于树的模型，将文档数据存储到一个由节点构成的层次结构中。程序可快速访问数据，因为所有文档数据都在内存中。DOM 还提供了相应的机制来增删节点，从而简化了程序对 XML 文档的修改。

基于 SAX 的解析器在遇到标记时，要调用“侦听器方法”。使用这种基于事件的模型，以 SAX 为

基础的解析器不会创建一个树结构来存储 XML 文档数据。相反,在找到数据时,解析器会将数据传给应用程序。相较于基于 DOM 的解析器,这样可获得更好的性能,内存开销也较小。事实上,许多 DOM 解析器都在幕后使用 SAX 解析器,以便从文档中获取数据,并在内存中生成 DOM 树。许多程序员都认为,使用 DOM 树结构,可以更容易地遍历及处理 XML 文档。因此,程序常常用 SAX 解析器读取不会由程序修改的 XML 文档。

性能提示 15.1 处理大型 XML 文档时,基于 SAX 的解析通常比基于 DOM 的解析更有效。尤其重要的是,基于 SAX 的解析器不会将整个 XML 文档载入内存。

性能提示 15.2 如果文档只需解析一次,那么最有效的就是基于 SAX 的解析。

性能提示 15.3 程序要从文档快速获取信息时,DOM 解析通常比 SAX 解析更有效。

性能提示 15.4 要求节省内存的程序通常使用基于 SAX 的解析器。

软件工程知识 15.2 尽管 SAX 已经获得了广泛的工业支持,但 XML-DEV 邮件列表的成员在开发 SAX 时是独立于 W3C 的。DOM 是正式的 W3C 推荐规范。

15.6 文档类型定义 (DTD)、架构和验证

本节介绍“文档类型定义”(DTD)和架构(Schema),即指定了 XML 文档结构的文档(包括哪些元素是允许的,一个元素可以有什么属性等等)。提供了一个 DTD 或者架构文档后,有的解析器(名为验证解析器¹⁾)会读取 DTD 或架构,并据此检查 XML 文档的结构。如 XML 文档符合 DTD 或架构的要求,就认为 XML 文档是“有效”的。如解析器不能根据 DTD 或架构来验证文档,就称为“非验证解析器”。如 XML 解析器(验证或非验证)能处理一个不引用 DTD 或架构的 XML 文档,就认为该 XML 文档是“良构”的,也就是符合语法规则的。按照定义,有效 XML 文档也是一个良构 XML 文档。如果文档不是良构的,解析器会产生一个错误。

软件工程知识 15.3 在 B2B(商家到商家)事务处理和关键任务处理系统中,DTD 和架构文档是 XML 文档的重要组成部分。

软件工程知识 15.4 由于 XML 文档内容可采用多种方式构造,所以应用程序不能确定它收到的文档数据是完整的,是丢失数据的,还是顺序正确的。DTD 和架构解决了这个问题,它们提供了全面的方式来描述文档的内容。应用程序可使用一个 DTD 或者架构文档对文档内容执行有效性验证。

15.6.1 DTD 文档

文档类型定义(DTD)为 XML 文档提供了类型检查途径,所以能验证其“有效性”(确定元素包含正确属性,元素处于正确顺序等等)。DTD 使用 EBNF(Extended Backus-Naur Form)语法来描述 XML 文档内容。XML 解析器需要附加的功能才能识别 EBNF 语法,因其并非 XML 语法。尽管 DTD 是可选的,但建议用它们确保文档一致性。图 15.7 的 DTD 定义了一个规则集(即语法),用于对图 15.8 的商务信函文档进行结构化。

```
1 <!-- Fig. 15.7: letter.dtd -->
2 <!-- DTD document for letter.xml. -->
3
4 <!ELEMENT letter ( contact+, salutation, paragraph+,
```

¹ 许多 DOM 解析器和 SAX 解析器都是验证解析器。请查阅您的解析器文档,了解它是否是验证解析器。

```

5   closing, signature )>
6
7   <!ELEMENT contact ( name, address1, address2, city, state,
8     zip, phone, flag )>
9   <!ATTLIST contact type CDATA #IMPLIED>
10
11  <!ELEMENT name ( #PCDATA )>
12  <!ELEMENT address1 ( #PCDATA )>
13  <!ELEMENT address2 ( #PCDATA )>
14  <!ELEMENT city ( #PCDATA )>
15  <!ELEMENT state ( #PCDATA )>
16  <!ELEMENT zip ( #PCDATA )>
17  <!ELEMENT phone ( #PCDATA )>
18  <!ELEMENT flag EMPTY>
19  <!ATTLIST flag gender (M | F) "M">
20
21  <!ELEMENT salutation ( #PCDATA )>
22  <!ELEMENT closing ( #PCDATA )>
23  <!ELEMENT paragraph ( #PCDATA )>
24  <!ELEMENT signature ( #PCDATA )>

```

图 15.7 用于商务信函的 DTD

移植性提示 15.2 DTD 可使不同程序生成的 XML 文档保持一致。

第 4 行使用“元素类型声明”ELEMENT 为 letter 元素定义规则。在本例中，letter 包含一个或多个 contact 元素、一个 salutation 元素、一个或多个 paragraph 元素、一个 closing 元素以及一个 signature 元素，而且必须按这个顺序排列。加号(+)是“出现次数指示符”，它表明元素必须出现一次或者多次。其他指示符包括星号(*)，表明一个可选元素可出现任意次数；以及问号(?)，表明一个可选元素最多只能出现一次。如省略出现次数指示符，就默认为刚好出现一次。

contact 元素定义(第 7 行)指定它按顺序包括 name, address1, address2, city, state, zip, phone 以及 flag 元素。而且每个元素都刚好出现一次。

第 9 行使用元素类型声明 ATTLIST 为 contact 元素定义一项属性(即 type)。关键字 #IMPLIED 表明，如解析器发现一个没有 type 属性的 contact 元素，应用程序可以提供一个值，或者忽略缺失的属性。缺少 type 属性，并不能使文档无效。其他类型的默认值包括 #REQUIRED 和 #FIXED。前者指出属性必须在文档中存在，后者指出必须总是为属性(如果有的话)指派一个特定的值。例如，

```
<!ATTLIST address zip #FIXED "01757">
```

表明必须为 zip 属性使用值 01757；否则，文档就是无效的。如果属性不存在，解析器会默认使用在 ATTLIST 声明中指定的固定值。CDATA 标记指出，type 属性中包含的文本不由解析器进行处理，而是原样传给应用程序。

软件工程知识 15.5 DTD 语法不能描述一个元素(或属性)的数据类型。

标记 #PCDATA(第 11 行)指定元素中可存储“解析过的字符数据”(即文本)。解析过的字符数据不能包含标记。由于要在标记中使用，所以字符<和&必须替换成它们的“实体引用”(即<和&)。然而，字符&可与实体引用结合使用。

第 18 行定义名为 flag 的一个空元素。关键字 EMPTY 指出元素不能包含字符数据。空元素通常用于它们的属性。

常见编程错误 15.8 未由 DTD 明确定义的任何元素、属性或关系会导致文档无效。

XML 文档必须显式引用一个 DTD，图 15.8 是遵循 letter.dtd(图 15.7)的 XML 文档。

```

1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.8: letter2.xml -->

```

```

4  <!-- Business letter formatted with XML. -->
5
6  <!DOCTYPE letter SYSTEM "letter.dtd">
7
8  <letter>
9      <contact type = "from">
10         <name>Jane Doe</name>
11         <address1>Box 12345</address1>
12         <address2>15 Any Ave.</address2>
13         <city>Othertown</city>
14         <state>Otherstate</state>
15         <zip>67890</zip>
16         <phone>555-4321</phone>
17         <flag gender = "F" />
18     </contact>
19
20     <contact type = "to">
21         <name>John Doe</name>
22         <address1>123 Main St.</address1>
23         <address2></address2>
24         <city>Anytown</city>
25         <state>Anystate</state>
26         <zip>12345</zip>
27         <phone>555-1234</phone>
28         <flag gender = "M" />
29     </contact>
30
31     <salutation>Dear Sir:</salutation>
32
33     <paragraph>It is our privilege to inform you about our new
34     database managed with XML. This new system
35     allows you to reduce the load on your inventory list
36     server by having the client machine perform the work of
37     sorting and filtering the data.
38     </paragraph>
39
40     <paragraph>Please visit our Web site for availability
41     and pricing.
42     </paragraph>
43     <closing>Sincerely</closing>
44     <signature>Ms. Doe</signature>
45 </letter>

```

图 15.8 引用相应 DTD 的 XML 文档

这个 XML 文档与图 15.3 所示文档相似。第 6 行引用一个 DTD 文件。该标记由 3 部分构成：要应用 DTD 的根元素名称（第 8 行的 `letter`）；关键字 `SYSTEM`（本例是指示一个“外部 DTD”，即在单独文件中定义的 DTD）；以及 DTD 的名称和位置（即当前目录中的 `letter.dtd`）。尽管可任意选择文件扩展名，但 DTD 文档通常采用 `.dtd` 扩展名。

许多工具（大部分免费）都可检查文档是否遵循 DTD 和架构（Schema，后文马上要讨论）。图 15.9 展示了使用 Microsoft XML Validator 参照 `letter.dtd` 对 `letter2.xml` 进行验证的结果。Microsoft XML Validator 的下载地址是：msdn.microsoft.com/downloads/samples/Internet/xml/xml_validator/sample.asp。要了解其他验证工具，请访问 www.w3.org/XML/Schema.html。

Microsoft XML Validator 既可参照本地 DTD 来验证 XML 文档，但您也可将文档上传到 XML Validator 网站进行远程验证。在本例中，`letter2.xml` 和 `letter.dtd` 位于 `/pythonhttp1/pythonhttp1_examples/Ch15` 目录。这个 XML 文档（`letter2.xml`）是有效的，因为它遵循 `letter.dtd`。

XML 文档即使验证失败，仍有可能是良构文档。如果一个文档不遵循 DTD 或架构，Microsoft XML Validator 会显示一条错误消息。例如，图 15.8 的 DTD 指出 `contacts` 元素必须包含子元素 `name`。如果遗漏该元素，文档虽然是良构的，但却是无效的。在这种情况下，Microsoft XML Validator 会显示如图 15.10 所示的错误消息。

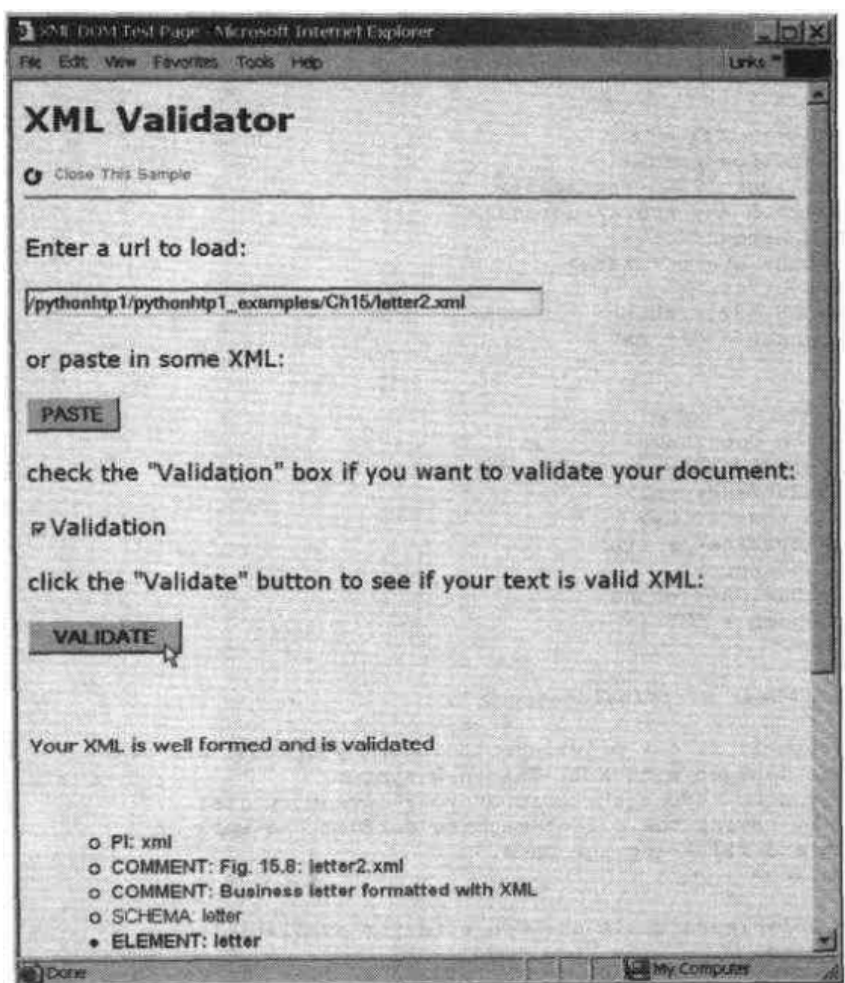


图 15.9 XML Validator 参照 DTD 来验证 XML 文档

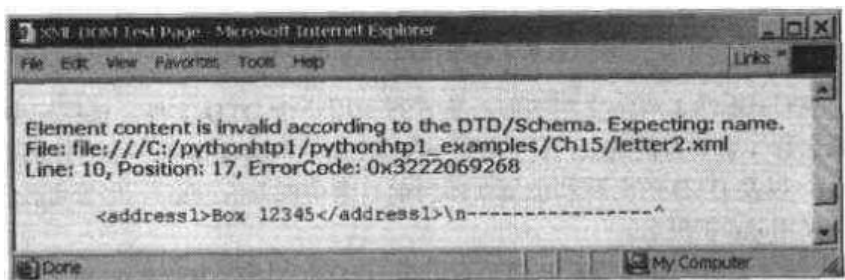


图 15.10 XML Validator 显示错误消息

15.6.2 W3C XML 架构文档

本节介绍 W3C XML 架构（详情请访问 www.w3.org/XML/Schema），它是 W3C 推荐规范（即稳定的发布版本，适合在业界推广使用）。XML 社区的许多开发者都认为 DTD 灵活性欠佳，不能满足当今的编程需要。例如，程序不能采取与 XML 文档相同的方式来处理 DTD（例如搜索，或者转换成 XHTML 等不同的表示形式），因为 DTD 本身不是 XML 文档。正是因为存在这些限制以及另一些问题，才促成了架构的问世。

和 DTD 不同，架构不使用 EBNF 语法。相反，架构使用 XML 语法，而且本质就是 XML 文档，可采取程序化的方式进行处理。类似 DTD，架构也需要验证解析器。在不远的将来，架构有望全面取代

DTD, 成为描述 XML 文档结构的主要手段。

DTD 描述的是 XML 文档的结构, 而非那个文档的元素内容。例如:

```
<quantity>5</quantity>
```

它包含字符数据。如果包含元素 `quantity` 的文档引用了一个 DTD, XML 解析器能够验证文档, 判断该元素是否真的包含 PCDATA 内容。然而, 解析器不能验证内容是不是数字; DTD 并不具备这方面的能力。所以令人遗憾的是, 像下面这样的标记:

```
<quantity>hello</quantity>
```

会被误认为有效。如果应用程序使用了包含该标记的 XML 文档, 必须检测元素 `quantity` 中的数据是不是数字。如果数据不是数字, 还要采取相应的行动。

XML 架构允许架构的作者指定元素 `quantity` 的数据必须为数字。解析器参照这个 Schema 对 XML 文档进行验证时, 可明确地判断出 5 符合要求, 但 `hello` 不符合。如 XML 文档符合一个架构文档的要求, 就认为它是“架构有效”的; 如果不符合要求, 就认为它是无效的。

本节参照 W3C XML 架构, 使用 XSV (XML Schema Validator, XML 架构验证程序) 来验证 XML 文档。要在线使用 XSV, 请访问 www.w3.org/2000/09/webdata/xsv, 输入要验证的 XML 文件名, 再单击 Upload and Get Results (上传并获取结果) 按钮。要想下载 XSV, 请访问 www.ltg.ed.ac.uk/~ht/xsv-status.html。

软件工程知识 15.6 许多组织和个人都在积极地为大范围的应用程序 (比如金融业务、医药处方等等) 创建 DTD 和架构。这些集合统称为 Repositories, 通常可从 Web 免费下载 (比如 www.dtd.com)。

图 15.11 展示了“架构有效”的 XML 文档 (`book.xml`), 图 15.12 展示了 W3C XML 架构文档 (`book.xsd`), 它定义了 `book.xml` 的结构。W3C XML 架构通常采用 `.xsd` 扩展名, 尽管这并非必需的。图 15.11 显示了参照 `book.xsd` 架构对 `book.xml` 进行验证的结果。注意, 输出结果是 XML, `outcome='success'` 和 `schemaErrors='0'` 属性表明 `book.xml` 是有效的。

```
1 <?xml version = "1.0"?>
2
3 <!-- Fig. 15.11: book.xml -->
4 <!-- Document that conforms to a W3C XML Schema. -->
5
6 <deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
7   <book>
8     <title>e-Business and e-Commerce How to Program</title>
9   </book>
10  <book>
11    <title>Python How to Program</title>
12  </book>
13 </deitel:books>
```

```
C:\Program Files\XSV>xsv /pythonhtml_examples/ch15/Schema/book.xml
/pythonhtml_examples/ch15/Schema/book.xsd

<?xml version='1.0'?>
<xsv docElt='{http://www.deitel.com/booklist}books'
instanceAssessed='true' instanceErrors='0'
rootType='{http://www.deitel.com/booklist}:BooksType'
schemaDocs='/pythonhtml_examples/ch15/Schema/book.xsd' schemaErrors='0'
target='file:/pythonhtml_examples/ch15/Schema/book.xml' validation='strict' version='XSV
1.203.2.37/1.106.2.19 of 2001/11/29 11:00:00' xmlns='http://www.w3.org/2000/05/xsv'>
<schemaDocAttempt URI='file:/pythonhtml_examples/ch15/Schema/book.xsd' outcome='success'
source='command line'/>
</xsv>
```

图 15.11 遵循 W3C XML 架构的 XML 文档


```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 15.12: book.xsd -->
4 <!-- Simple W3C XML Schema document. -->
5
6 <xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
7   xmlns:deitel = "http://www.deitel.com/booklist"
8   targetNamespace = "http://www.deitel.com/booklist">
9
10  <xsd:element name = "books" type = "deitel:BooksType" />
11
12  <xsd:complexType name = "BooksType">
13    <xsd:sequence>
14      <xsd:element name = "book" type = "deitel:BookType"
15        minOccurs = "1" maxOccurs = "unbounded" />
16    </xsd:sequence>
17  </xsd:complexType>
18
19  <xsd:complexType name = "BookType">
20    <xsd:sequence>
21      <xsd:element name = "title" type = "xsd:string" />
22    </xsd:sequence>
23  </xsd:complexType>
24
25 </xsd:schema>

```

图 15.12 book.xml 遵循的 XSD 架构文档

W3C XML 架构使用命名空间 URI <http://www.w3.org/2001/XMLSchema>，而且通常使用命名空间前缀 `xsd`（图 15.12 的第 6 行）。根元素 `schema` 包含定义了 XML 文档结构的元素。第 7 行将 URI <http://www.deitel.com/booklist> 绑定到命名空间前缀 `deitel`。第 8 行指定了 `targetNamespace`，它是该架构所定义的元素和属性的命名空间。

良好编程习惯 15.1 按照约定，如果引用 URI <http://www.w3.org/2001/XMLSchema>，W3C XML 架构作者要使用命名空间前缀 `xsd`。

在 W3C XML 架构中，`element` 元素（第 10 行）定义一个元素。`name` 和 `type` 属性分别指定元素名称和数据类型。在这个例子中，元素名称是 `books`，数据类型是 `deitel:BooksType`。包含属性或子元素的任何一个元素（例如 `books`）都必须定义一个“复杂类型”，它定义了每个子元素。例如，`deitel:BooksType` 类型（第 12~17 行）就是一个复杂类型。我们为 `BooksType` 附加 `deitel` 前缀，因为它是我们创建的一个复杂类型，而不是现有的 W3C XML 架构数据类型。

第 12~17 行使用 `complexType` 元素定义一个元素类型，其中包含一个子元素，名为 `book`。由于 `book` 包含一个子元素，所以它的类型肯定是一种复杂类型（`BookType`）。`minOccurs` 属性指出 `books` 至少要包含一个 `book` 元素。`maxOccurs` 属性的值为 `unbounded`（第 14 行），它指出 `books` 可以有任意数量的 `book` 子元素。`sequence` 元素指定了元素在复杂类型中的顺序。

第 19~23 行定义了复杂类型 `BookType`。第 21 行定义类型为 `xsd:string` 的 `title` 元素。如果元素具有 `xsd:string` 这样的简单类型，就不能包含属性和子元素。W3C XML 架构提供了大量数据类型，比如用于日期的 `xsd:date`，用于整数的 `xsd:int`，用于浮点数的 `xsd:double` 以及用于时间的 `xsd:time`。

图 15.12 的架构指出，每个 `book` 元素都必须包含子元素 `title`。如遗漏该元素，文档虽然是良构的，但却是无效的。如果从图 15.11 移除第 8 行，XSV 会显示如图 15.13 所示的错误消息。

```

C:\PROGRA~1\XSV>xsv /pythonhttp1/pythonhttp1_examples/Ch15/Schema/book.xml
/pythonhttp1/pythonhttp1_examples/Ch15/Schema/book.xsd

<?xml version='1.0'?>
<xsv docElt='{http://www.deitel.com/booklist}books' instanceAssessed='true'
instanceErrors='1' rootType='{http://www.deitel.com/booklist}:BooksType'
schemaDocs='/pythonhttp1/pythonhttp1_examples/Ch15/Schema/book.xsd' schemaErrors='0'
target='file:/pythonhttp1/pythonhttp1_examples/Ch15/Schema/book.xml' validation='strict'

```



```

version='XSV 1.203.2.37/1.106.2.19 of 2001/11/29 11:00:00'
xmlns='http://www.w3.org/2000/05/xsv'>
<schemaDocAttempt URI='file:/pythonhtml/pythonhtml_examples/Ch15/Schema/book.xsd'
outcome='success' source='command line'/>
<invalid char='4' code='cvc-complex-type.1.2.4' line='8'
resource='file:/pythonhtml/pythonhtml_examples/Ch15/Schema/book.xml'>content of book is not
allowed to end here (1), expecting ['{None}:title']>
<fsm>
<node id='1'>
<edge dest='2' label='{None}:title'/>
</node>
<node final='true' id='2'/>
</fsm></invalid>
</xsv>

```

图 15.13 不遵循 W3C XML 架构的 XML 文档

15.7 XML 词汇表

XML 允许作者自行创建标记,以便精确描述数据。各个学术领域的许多个人和组织创建了大量 XML 词汇表来结构化数据。其中一些词汇表包括 MathML (数学标记语言)、可扩展矢量图形 (SVG)、无线标记语言 (WML)、可扩展商业报表语言 (XBRL)、可扩展用户界面语言 (XUL) 以及 VoiceXML。XML 词汇表的另两个例子是 W3C XML 架构和 15.8 节介绍的可扩展样式表语言 (XSL)。后文将描述 MathML、化学标记语言 (CML) 和其他 XML 词汇表。

15.7.1 MathML

不久之前,计算机通常需要专业软件包 (比如 TeX 和 LaTeX) 才能显示复杂的数学表达式。本节要介绍 MathML, 它由 W3C 开发,用于描述数学符号及表达式。可解析和呈现 MathML 的一个应用程序是 W3C 的 Amaya 浏览器/编辑器,可到以下网址免费下载:

www.w3.org/Amaya/User/BinDist.html

这个网页包含了针对 Windows 95/98/NT/2000、Linux 和 Solaris 平台的下载链接。在 W3C 网站,还提供了 Amaya 的用户文档及安装指南。

MathML 标记描述要显示的数学表达式。图 15.4 使用 MathML 标记一个简单表达式。注意本节提供的示范输出,它展示了具有 MathML 功能的应用程序应如何呈现标记。

```

1 <?xml version="1.0"?>
2
3 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
4   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
5
6 <!-- Fig. 15.14: mathml1.html -->
7 <!-- Simple MathML. -->
8
9 <html xmlns = "http://www.w3.org/1999/xhtml">
10
11   <head><title>Simple MathML Example</title></head>
12
13   <body>
14
15     <math xmlns = "http://www.w3.org/1998/Math/MathML">
16
17       <mrow>
18         <mn>2</mn>
19         <mo>+</mo>
20         <mn>3</mn>

```

```

21         <mo>=</mo>
22         <mn>5</mn>
23     </mrow>
24
25 </math>
26
27 </body>

```

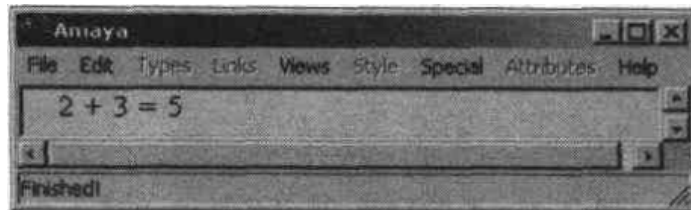


图 15.14 用 MathML 标记的表达式

使用 `math` 元素和默认命名空间 <http://www.w3.org/1998/Math/MathML> (第 15 行), 我们将 MathML 内容嵌入一个 XHTML 文档。`mrow` 元素 (第 17 行) 是一个表达式容器元素, 其中包括多个元素。在这个例子中, `mrow` 元素总共包含 5 个子元素。`mn` 元素 (第 18 行) 标记一个数字。`mo` 元素 (第 19 行) 标记一个运算符 (例如+)。利用这个标记, 我们定义了表达式 $2 + 3 = 5$, 支持 MathML 的软件可以显示这种表达式。

现在讨论如何用 MathML 标记一个代数方程式, 它使用了指数和算术运算符, 如图 15.15 所示。

```

1 <?xml version="1.0"?>
2
3 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
4   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
5
6 <!-- Fig. 15.15: mathml2.html -->
7 <!-- Simple MathML. -->
8
9 <html xmlns = "http://www.w3.org/1999/xhtml">
10
11   <head><title>Algebraic MathML Example</title></head>
12
13   <body>
14
15     <math xmlns = "http://www.w3.org/1998/Math/MathML">
16       <mrow>
17
18         <mrow>
19           <mn>3</mn>
20           <mo>&InvisibleTimes;</mo>
21
22           <msup>
23             <mi>x</mi>
24             <mn>2</mn>
25           </msup>
26
27         </mrow>
28
29         <mo>+</mo>
30         <mi>x</mi>
31         <mo>-</mo>
32
33         <mfrac>
34           <mn>2</mn>
35           <mi>x</mi>
36         </mfrac>
37
38         <mo>=</mo>
39         <mn>0</mn>
40
41       </mrow>

```

```

42     </math>
43
44     </body>
45 </html>

```

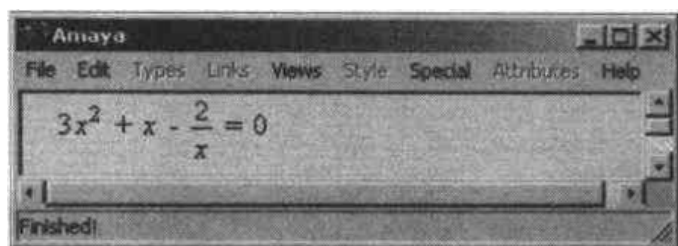


图 15.15 用 MathML 标记的代数方程式

mrow 元素行为类似于圆括号，可用它对相关元素进行正确的分组。第 20 行使用实体引用 ⁢ 标记一个没有符号表示的乘法运算（即 3 和 x 之间没有乘号）。对于指数，第 22 行使用 msup 元素来表示一个上标。msup 元素有两个子元素，也就是基数和指数。类似地，msub 元素表示一个下标。为显示 x 这样的变量，第 23 行使用标识符元素 mi。

为显示分数，第 33 行使用 mfrac 元素。第 34~35 行指定了分数的分子和分母。如果分子或分母包含多个元素，就必须把它们嵌套到一个 mrow 元素中。

图 15.16 标记一个微积分表达式，其中包括积分和平方根符号。

```

1  <?xml version="1.0"?>
2
3  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
4    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
5
6  <!-- Fig. 15.16: mathml3.html      -->
7  <!-- Calculus example using MathML. -->
8
9  <html xmlns = "http://www.w3.org/1999/xhtml">
10
11    <head><title>Calculus MathML Example</title></head>
12
13    <body>
14
15      <math xmlns = "http://www.w3.org/1998/Math/MathML">
16        <mrow>
17          <msubsup>
18
19            <mo>&Integral;</mo>
20            <mn>0</mn>
21
22            <mrow>
23              <mn>1</mn>
24              <mo>-</mo>
25              <mi>y</mi>
26            </mrow>
27          </msubsup>
28
29          <msqrt>
30            <mrow>
31
32              <mn>4</mn>
33              <mo>&InvisibleTimes;</mo>
34
35              <msup>
36                <mi>x</mi>
37                <mn>2</mn>
38              </msup>
39            </mrow>
40            <mo>+</mo>
41            <mi>y</mi>
42

```

```

43
44         </mrow>
45     </msqrt>
46
47     <mo>&delta;</mo>
48     <mi>x</mi>
49 </mrow>
50 </math>
51 </body>
52 </html>

```

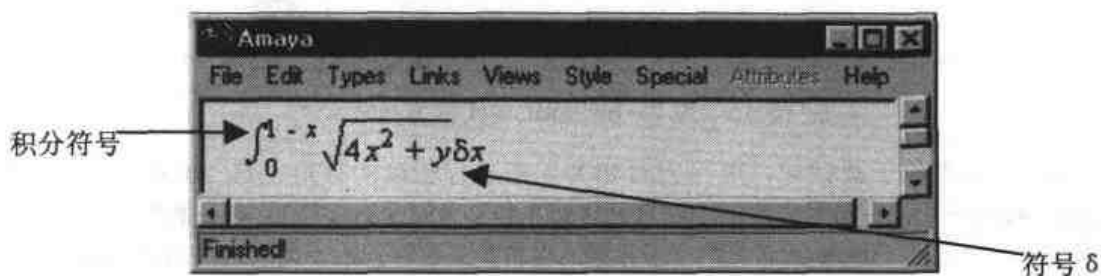


图 15.16 用 MathML 标记微积分表达式

第 19 行的实体引用 `∫` 代表积分符号，而 `msubsup` 元素（第 17 行）指定上标和下标。元素 `mo` 标记积分运算符。`msubsup` 元素要求 3 个子元素——一个运算符（例如积分实体引用）、下标表达式（第 20 行）以及上标表达式（第 22~26 行）。`mn` 元素（第 20 行）标记代表下标的数字（即 0）。`mrow` 元素则标记了代表上标表达式（即 $1-x$ ）。

`msqrt` 元素（第 30~45 行）代表一个平方根表达式。第 31 行使用 `mrow` 元素对平方根中包含的表达式进行分组。第 47 行用实体引用 `δ` 表示一个 δ 符号。 δ 是运算符，所以第 47 行将这个实体引用放到元素 `mo` 中。要了解详情 MathML 的其他运算和符号，请访问 www.w3.org/Math。

15.7.2 化学标记语言（CML）

化学标记语言（Chemical Markup Language, CML）是用于表示分子和化学信息的 XML 词汇表。尽管许多读者不具备完全理解本节的例子所需的化学知识，但由于 CML 完美展现了 XML 的用途，所以我们仍然包括了这个例子，向读者展示 XML 的强大功能。文档作者可使用 Jumbo 浏览器^①来编辑和查看 CML。它的网址是 www.xml-cml.org。图 15.17 是用 CML 标记的一个氨分子。

```

1 <?jumbo:namespace ns = "http://www.xml-cml.org"
2   prefix = "C" java = "jumbo.cmlxml.*Node" ?>
3
4 <!-- Fig. 15.17: ammonia.xml -->
5 <!-- Structure of ammonia. -->
6
7 <C:molecule id = "Ammonia">
8
9   <C:atomArray builtin = "elsym">
10     N H H H
11   </C:atomArray>
12
13   <C:atomArray builtin = "x2" type = "float">
14     1.5 0.0 1.5 3.0
15   </C:atomArray>
16
17   <C:atomArray builtin = "y2" type = "float">
18     1.5 1.5 0.0 1.5
19   </C:atomArray>
20

```

^① 写作本书时，Jumbo 还不允许用户载入文档以进行呈现。为方便陈述，我们自行绘制了图 15.17 的分子结构图。

```

21  <C:bondArray builtin = "atid1">
22    1 1 1
23  </C:bondArray>
24
25  <C:bondArray builtin = "atid2">
26    2 3 4
27  </C:bondArray>
28
29  <C:bondArray builtin = "order" type = "integer">
30    1 1 1
31  </C:bondArray>
32
33 </C:molecule>

```

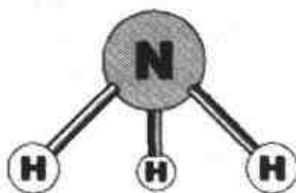


图 15.17 用 CML 标记表示的氮分子

第 1~2 行是一个“处理指令”(PI), 其中包含要嵌入 XML 文档的应用程序特有的信息。字符<?和?>对处理指令进行定界。第 1~2 行的处理指令为 Jumbo 浏览器提供了应用程序特有的信息。处理指令包括一个“PI 目标”(例如 jumbo:namespace) 以及一个 PI 值(例如 ns = "http://www.xml-cml.org" prefix = "C" java = "jumbo.cmlxml.*Node")。

移植性提示 15.3 处理指令允许文档作者在不影响文档移植性的情况下, 向 XML 文档嵌入应用程序特有的信息。

第 7 行使用 molecule 元素定义一个氮分子。id 属性把这个分子标识为 Ammonia。第 9~11 行使用 atomArray 元素和 builtin 属性指定分子的原子。氮分子包括 1 个氮原子和 3 个氢原子。

第 13~15 行显示了 atomArray 元素, 它将 builtin 属性值设为 x2, 将 type 设为 float。这指明元素中包含一个浮点数列表, 每个数字都指定了一个原子的 x 坐标。第一个值(1.5)是第一个原子(氮)的 x 坐标, 第二个值(0.0)是第二个原子(第一个氢原子)的 x 坐标, 以此类推。

第 17~19 行显示了 atomArray 元素, 它将 builtin 属性值设为 y2, 将 type 设为 float。这指明元素包含一个 y 坐标值列表。第一个值(1.5)是第一个原子(氮)的 y 坐标, 第二个值(1.5)是第二个原子(第一个氢原子)的 y 坐标, 以此类推。第 21~23 行显示了 bondArray 元素, 它将 builtin 属性值设为 atid1。bondArray 元素定义了原子之间的键。该元素的 builtin 值为 atid1, 所以这个元素指定的值构成了一对原子中的第一个原子。由于要定义 3 个键, 所以指定了 3 个值。针对每个值, 都指定 atomArray 中的第一个原子, 即氮原子。第 25~27 行显示了 bondArray 元素, 它将 builtin 属性值设为 atid2。这个元素的值构成了一对原子中的第二个原子, 并指定了 3 个氢原子。

第 29~31 行显示了 bondArray 元素, 它将 builtin 属性值设为 order, 将 type 设为 integer。该元素的值代表原子对之间的键数。所以, 氮原子和第一个氢原子之间的键是单键, 氮原子和第二个氢原子之间的键是单键, 氮原子和第三个氢原子之间的键也是单键。

15.7.3 其他 XML 词汇表

从 XML 派生出了数百种 XML 词汇表, 人们每天都能为 XML 找到新用途。图 15.18 进行了总结。

词汇表	说明
VoiceXML	VoiceXML 论坛由 AT&T, IBM, Lucent 和 Motorola 联合建立, 致力于

词汇表	说明
	VoiceXML 标准的制定。在 VoiceXML 的帮助下,人们可通过电话、PDA (个人数字助理)或者桌面计算机实现人机交互式语音通信。IBM 的 VoiceXML SDK 可用于处理 VoiceXML 文档。有关 VoiceXML 的详情,请访问 www.voicexml.org
同步多媒体集成语言 (Synchronous Multimedia Integration Language, SMIL)	SMIL 是一种用于多媒体演示的 XML 词汇表。W3C 是 SMIL 的主要开发者。其他公司也参与了这个项目的部分开发工作。有关 SMIL 的详情,请访问 www.w3.org/AudioVideo
研究信息交换标记语言 (Research Information Exchange Markup Language, RIXML)	RIXML 由经纪人商家联盟开发,用于标记投资数据。有关 RIXML 的详情,请访问 www.rxml.org
ComicsML	由 Jason Macintosh 开发的一种语言,用于标记电脑漫画。要更多地了解 ComicsML,请访问 www.jmac.org/projects/comics_ml
地理标记语言 (Geography Markup Language, GML)	OpenGIS 开发了 GML 以描述地理信息。有关 GML 的详情,请访问 www.opengis.org
可扩展用户界面语言 (Extensible User Interface Language, XUL)	Mozilla 项目创建了 XUL,可采取独立于平台的方式描述图形用户界面。要想了解详细信息,请访问 www.mozilla.org/xpfe/languageSpec.html

图 15.18 XML 词汇表

15.8 可扩展样式表语言 (XSL)^①

可扩展样式表语言 (Extensible Stylesheet Language, XSL) 是一种 XML 词汇表,用于格式化 XML。本节讨论了 XSL 的 XSLT 部分,即“XSL 转换”(XSLT),它可根据 XML 文档创建以格式化好的文本为基础的文档。这个创建过程称为“转换”,其中牵涉到两个树结构,即源树(要转换的 XML 文档)和结果树(转换结果,例如 XHTML^②)。完成转换之后,源树不会发生变动。

要执行转换,需要使用 XSLT 处理器。流行的 XSLT 处理器包括微软的 msxml、Apache Software Foundation 的 Xalan 2 和 Python 的 4XSLT 包(第 16 章将使用这个包)。图 15.19 的 XML 文档由 msxml 转换成一个 XHTML 文档,这是利用图 15.20 的 XSLT 文档来进行的。

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 15.19: sorting.xml -->
4 <!-- XML document containing book information. -->
5
6 <?xml:stylesheet type = "text/xsl" href = "sorting.xsl"?>
7
8 <book isbn = "999-99999-9-X">
9   <title>Mary's XML Primer</title>
10
11   <author>
12     <firstName>Mary</firstName>
13     <lastName>White</lastName>
14   </author>
15
16   <chapters>
17     <frontMatter>
18       <preface pages = "2" />
19       <contents pages = "5" />
20       <illustrations pages = "4" />
21     </frontMatter>
22

```

^① 本节的例子需要运行 msxml 3.0 或更高版本。要想了解有关 msxml 3.0 的下载和安装的详情,请访问 www.deitel.com。

^② XHTML 是 W3C 推荐规范,用于取代 HTML 以标识 Web 内容。

```

23     <chapter number = "3" pages = "44">
24         Advanced XML</chapter>
25     <chapter number = "2" pages = "35">
26         Intermediate XML</chapter>
27     <appendix number = "B" pages = "26">
28         Parsers and Tools</appendix>
29     <appendix number = "A" pages = "7">
30         Entities</appendix>
31     <chapter number = "1" pages = "28">
32         XML Fundamentals</chapter>
33 </chapters>
34
35 <media type = "CD" />
36 </book>

```

图 15.19 包含书籍信息的 XML 文档

第 6 行是 IE 专用的处理指令, 指定了要应用于该 XML 文档的 XSLT 文档的位置。图 15.20 展示了能将 sorting.xml 转换成 XHTML 格式的 XSLT 文档 (sorting.xsl)。

性能提示 15.5 在客户端上利用 Internet Explorer 来处理 XSLT 文档, 可显著节省服务器资源, 因为服务器不必亲自为多个客户处理 XSLT 文档。

```

1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.20: sorting.xsl -->
4  <!-- Transformation of book information into XHTML. -->
5
6  <xsl:stylesheet version = "1.0"
7      xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
8
9      <!-- write XML declaration and DOCTYPE DTD information -->
10     <xsl:output method = "xml" omit-xml-declaration = "no"
11         doctype-system =
12             "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
13         doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
14
15     <!-- match document root -->
16     <xsl:template match = "/">
17         <html xmlns = "http://www.w3.org/1999/xhtml">
18             <xsl:apply-templates />
19         </html>
20     </xsl:template>
21
22     <!-- match book -->
23     <xsl:template match = "book">
24         <head>
25             <title>ISBN <xsl:value-of select = "@isbn" /> -
26             <xsl:value-of select = "title" /></title>
27         </head>
28
29         <body>
30             <h1 style = "color: blue">
31                 <xsl:value-of select = "title"/></h1>
32
33             <h2 style = "color: blue">by <xsl:value-of
34                 select = "author/lastName" />,
35                 <xsl:value-of select = "author/firstName" /></h2>
36
37             <table style =
38                 "border-style: groove; background-color: wheat">
39
40                 <xsl:for-each select = "chapters/frontMatter/*">
41                     <tr>
42                         <td style = "text-align: right">
43                             <xsl:value-of select = "name()" />
44                         </td>
45
46                         <td>

```

```

47         ( <xsl:value-of select = "@pages" /> pages )
48     </td>
49 </tr>
50 </xsl:for-each>
51
52 <xsl:for-each select = "chapters/chapter">
53     <xsl:sort select = "@number" data-type = "number"
54         order = "ascending" />
55     <tr>
56         <td style = "text-align: right">
57             Chapter <xsl:value-of select = "@number" />
58         </td>
59
60         <td>
61             ( <xsl:value-of select = "@pages" /> pages )
62         </td>
63     </tr>
64 </xsl:for-each>
65
66 <xsl:for-each select = "chapters/appendix">
67     <xsl:sort select = "@number" data-type = "text"
68         order = "ascending" />
69     <tr>
70         <td style = "text-align: right">
71             Appendix <xsl:value-of select = "@number" />
72         </td>
73
74         <td>
75             ( <xsl:value-of select = "@pages" /> pages )
76         </td>
77     </tr>
78 </xsl:for-each>
79 </table>
80
81 <p style = "color: blue">Pages:
82     <xsl:variable name = "pagecount"
83         select = "sum(chapters/*/@pages)" />
84     <xsl:value-of select = "$pagecount" />
85 <br />Media Type:
86     <xsl:value-of select = "media/@type" /></p>
87 </body>
88 </xsl:template>
89
90 </xsl:stylesheet>

```

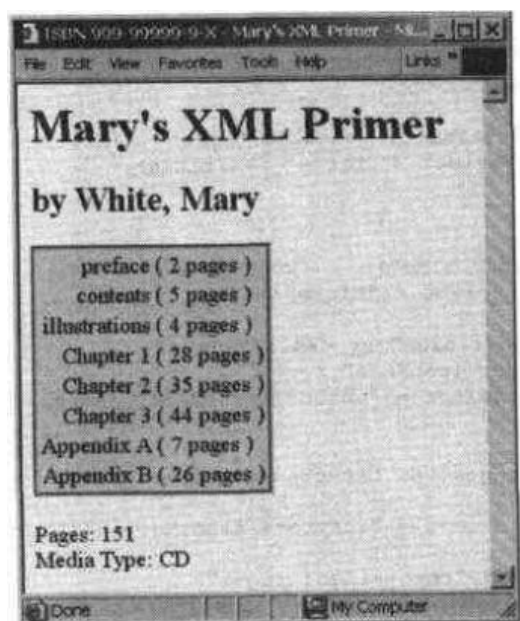


图 15.20 可将 sorting.xml 转换成 XHTML 的 XSLT 文档

图 15.20 的第 1 行包含 XML 声明。之所以有这一行,是因为 XSLT 文档本身就是 XML 文档。第 6 行是 `xsl:stylesheet` 根元素。`version` 属性指定该文档所遵循的 XSLT 的版本。定义了命名空间前缀 `xsl`,并绑定到 W3C 所定义的 XSLT URI。处理完毕后,第 11~13 行将文档类型声明写入结果树。`method` 属性被指派“xml”值,它表明 XML 要输出到结果树。`omit-xml-declaration` 属性值设为“no”,表明会将一个 XML 声明输出到结果树。`doctype-system` 和 `doctype-public` 属性包含要输出到结果树的 Doctype DTD 信息。

XSLT 文档包含一个或多个 `xsl:template` 元素,它指定 XSLT 处理器要将哪些信息输出到结果树。第 16 行的模板匹配源树的文档根。转换期间遇到文档根时,就会应用这个模板。另外,由这个元素标记的任何文本如果不在由 `xsl` 引用的命名空间中,那么会输出到结果树。在第 18 行,调用与“准备应用的文档根的子”匹配的所有模板。第 23 行指定一个与 `book` 元素相匹配的模板。

第 25~26 行为 XHTML 文档创建标题。我们用书籍的 ISBN 号(来自 `isbn` 属性)和 `title` 元素的内容来创建标题字符串 ISBN 999-99999-9-X - Mary's XML Primer。`xsl:value-of` 元素选择了 `book` 元素的 `isbn` 属性。

第 33~35 行创建一个标题元素,其中包含书籍作者。由于“背景节点”(即正在处理的当前节点)是 `book`,所以表达式 `author/lastName` 选择作者的姓氏,表达式 `author/firstName` 选择作者的名字。

第 40 行选择作为 `frontMatter` 元素的一个“子”的每一个元素(由星号指示)。第 43 行调用节点设置函数 `name` 来获取当前节点的元素名(如 `preface`)。当前节点是 `xsl:for-each` 指定的背景节点(第 40 行)。

第 53~54 行按章号对 `chapter` 进行升序排序。`select` 属性选择背景节点 `chapter` 的 `number` 属性的值。`data-type` 属性的值是“number”,它指定一个数值排序;`order` 属性指定了“ascending”(升序)顺序。第 67 行为 `data-type` 属性指派了“text”值。另外,可为 `order` 属性指派“descending”(降序)值。

第 82~83 行使用一个 XSLT 变量来存储书的页数,并把它输出到结果树。`name` 属性指定变量名称,`select` 属性为其指派一个值。`sum` 函数对所有 `page` 属性的值进行求和。`chapters` 和 `*` 之间的两个正斜杠指明:要搜索 `chapters` 的所有后代节点,从中查找包含 `pages` 属性的所有元素。

图 15.21 展示了 `msxml` 将 `sorting.xml` 应用于 `sorting.xml` 之后生成的 XHTML。下一章将使用几个 Python 的 XML 包,将 XSLT 样式表应用于 XML 文档。

注意 XHTML 文档包含一个 XML 声明,它有别于以前显示的声明。`encoding` 值指明文档使用的字符编码类型(也就是和字符对应的数值)。这个文档使用的是 UTF-8,它尤其适合基于 ASCII 的系统。UTF-8 是 XML 文档的默认编码。

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <html xmlns="http://www.w3.org/1999/xhtml">
6
7    <head>
8      <title>ISBN 999-99999-9-X - Mary's XML Primer</title>
9    </head>
10
11    <body>
12      <h1 style="color: blue">Mary's XML Primer</h1>
13      <h2 style="color: blue">by White, Mary</h2>
14      <table style="border-style: groove; background-color: wheat">
15
16        <tr>
17          <td style="text-align: right">preface</td>
18          <td>( 2 pages )</td>
19        </tr>
20
21        <tr>
22          <td style="text-align: right">contents</td>
23          <td>( 5 pages )</td>
24        </tr>
25
26        <tr>
27          <td style="text-align: right">illustrations</td>

```

```

28         <td>( 4 pages )</td>
29     </tr>
30
31     <tr>
32         <td style="text-align: right">Chapter 1</td>
33         <td>( 28 pages )</td>
34     </tr>
35
36     <tr>
37         <td style="text-align: right">Chapter 2</td>
38         <td>( 35 pages )</td>
39     </tr>
40
41     <tr>
42         <td style="text-align: right">Chapter 3</td>
43         <td>( 44 pages )</td>
44     </tr>
45
46     <tr>
47         <td style="text-align: right">Appendix A</td>
48         <td>( 7 pages )</td>
49     </tr>
50
51     <tr>
52         <td style="text-align: right">Appendix B</td>
53         <td>( 26 pages )</td>
54     </tr>
55
56 </table>
57
58 <p style="color: blue">Pages: 11<br />Media Type: CD</p>
59
60 </body>
61 </html>

```

图 15.21 msxml 将 sorting.xsl 应用于 sorting.xml 所生成的 XHTML

15.9 因特网和万维网资源

www.w3.org/xml

W3C（万维网协会）致力于开发通用协议，确保在 Web 上的互操作性。它们的 XML 主页提供了与活动安排、出版物、软件和讨论组有关的信息。请访问这个网站，了解 XML 的最新进展。

www.xml.org

提供有关 XML、DTD、架构（Schema）和命名空间的大量参考资料。还提供了 XML 行业新闻。

www.w3.org/style/XSL

提供与 XSL 有关的信息，包括 XSL 新特性、学习 XSL、支持 XSL 的工具、XSL 规范、FAQ、XSL 历史等等。

www.w3.org/TR

这是 W3C 的技术报告和出版物主页，可通过其中的链接访问 W3C 工作草案、提议规范、推荐规范等等。

xml.apache.org

Apache XML 网站提供与 XML 有关的大量资源，包括工具和下载。

www.xmlbooks.com

Charles Goldfarb 在此推荐了大量优秀的 XML 书籍。Charles Goldfarb 是 GML（常规标记语言）的设计者之一，SGML（XML 的父语言）正是在 GML 的基础上派生出来的。

wdvl.internet.com/Authoring/Languages/XML

“Web 开发者虚拟图书馆”的 XML 主页包括大量教程、FAQ、新闻和链接。

www.xml.com

提供有关 XML 的最新新闻和信息。另外还有会议列表以及大量网上 XML 资源链接（按主题和工具分类）。

msdn.microsoft.com/xml/default.asp

“MSDN 在线 XML 开发中心”，提供许多与 XML 有关的实用文章。其他特色内容包括“Ask the Experts”聊天区（和专家聊天）、示例和演示、新闻组等等。

www.oasis-open.org/cover/xml.html

SGML/XML 主页是一个内容丰富的信息资源，提供了大量 FAQ、在线资源、行业事件、演示、会议及教程链接。

www.gca.org/whats_xml/default.htm

GCA 网站提供 XML 术语表、书籍列表、对 XML 草案的简要说明以及到在线草案的链接。

www.xmlinfo.com

可通过这里提供的链接访问 XML 教程、推荐参考书列表、文档、论坛等等。

developer.netscape.com/tech/metadata/index.html

“XML 和 Metadata 开发者中心”提供同 XML 有关的许多演示、技术评论和新闻稿。

www.ucc.ie/xml

提供详尽的 XML FAQ，也可提交您自己的问题。

www.xml-cml.org

“化学标记语言”（CML）主页，包括 FAQ、文档、软件和 XML 链接。

第 16 章 Python 的 XML 处理

学习目标

- 用程序化的手段创建 XML 标记
- 用文档对象模型 (DOM) 处理 XML 文档
- 用 Simple API for XML (SAX) 从 XML 文档获取数据
- 创建一个基于 XML 消息的论坛

16.1 概述

第 15 章介绍了 XML 和各种 XML 相关技术。本章要演示 Python 应用程序和脚本如何处理 XML 文档。对 XML 的支持是通过大量免费的 Python 包和模块来实现的。本章重点讨论两个 Python 包: 4DOM 和 xml.sax。

本章将讨论如何程序化地生成 XML 内容, 介绍如何在以程序化的方式处理 XML 文档的数据, 进行基于 DOM 和 SAX 的解析。最后提供了一个案例分析, 它用 XML 标记论坛数据。

16.2 动态生成 XML 内容

Python 应用程序动态生成 XML 的过程类似于生成 XHTML 的过程。例如, 要从 Python 脚本中生成 XML, 可使用 print 语句或者 XSLT。

本节展示了一个简单的 Python 脚本, 它根据一个文本文件 (参见图 16.1) 中的数据来创建一个 XML 文档。XML 标记通过 print 语句发送到浏览器。16.4 节将介绍如何用更高级的技术来创建和处理 XML 文档。图 16.2 是将文本文件的数据标记成 XML 的 Python 脚本。注意在本例中, names.txt, fig16_02.py 和 contact_list.xml 等文件必须放到正确目录中, 使 Apache 能正确地提供服务。尤其要注意的是, names.txt 和 fig16_02.py 必须位于 Apache 的 cgi-bin 目录; contact_list.xml 必须位于 Apache 的 htdocs 目录下的 XML 子目录。正确的目录结构也将在图 16.19 进行展示。

```
1 O'Black, John
2 Green, Sue
3 Red, Bob
4 Blue, Mary
5 White, Mike
6 Brown, Jane
7 Gray, Bill
```

图 16.1 将用于图 16.2 的文本文件 names.txt

```
1 #!c:\Python\python.exe
2 # Fig. 16.2: fig16_02.py
3 # Marking up a text file's data as XML.
4
5 import sys
6
7 print "Content-type: text/xml\n"
8
9 # write XML declaration and processing instruction
10 print "<?xml version = '1.0'>"
11 <?xml:stylesheet type = "text/xsl"
12 href = "../XML/contact_list.xml">"
13
```

```

14 # open data file
15 try:
16     file = open( "names.txt", "r" )
17 except IOError:
18     sys.exit( "Error opening file" )
19
20 print "<contacts>" # write root element
21
22 # list of tuples: ( special character, entity reference )
23 replaceList = [ ( "&", "&amp;" ),
24                 ( "<", "&lt;" ),
25                 ( ">", "&gt;" ),
26                 ( "'", "&quot;" ),
27                 ( '"', "&apos;" ) ]
28
29 # replace special characters with entity references
30 for currentLine in file.readlines():
31
32     for oldValue, newValue in replaceList:
33         currentLine = currentLine.replace( oldValue, newValue )
34
35     # extract lastname and firstname
36     last, first = currentLine.split( " ", " )
37     first = first.strip() # remove carriage return
38
39     # write contact element
40     print "" <contact>"
41         <LastName>%s</LastName>
42         <FirstName>%s</FirstName>
43     </contact>" "" % ( last, first )
44
45 file.close()
46
47 print "</contacts>"

```

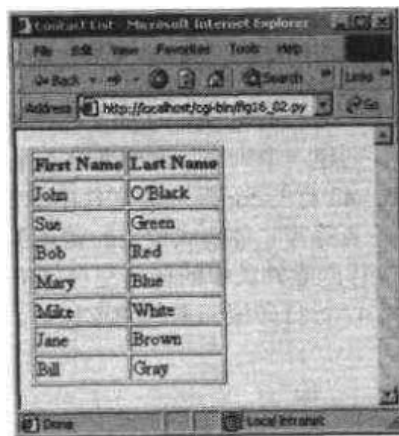


图 16.2 将文本文件的数据标记成 XML

第7行打印 HTTP 标头，将 MIME 类型设为 text/xml。第10~12行打印 XML 声明和用于 Internet Explorer 的一条“处理指令”。该处理指令引用了名为 contact_list.xsl 的 XSLT 样式表，如图 16.3 所示。

```

1 <?xml version = "1.0"?>
2 <!-- Fig. 16.3: contact_list.xsl -->
3 <!-- Formats a contact list -->
4
5 <xsl:stylesheet version = "1.0"
6     xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
7
8     <!-- match document root -->
9     <xsl:template match = "/">
10
11         <html xmlns = "http://www.w3.org/1999/xhtml">
12

```

```

13     <head>
14         <title>Contact List</title>
15     </head>
16
17     <body>
18         <table border = "1">
19
20             <thead>
21                 <tr>
22                     <th>First Name</th>
23                     <th>Last Name</th>
24                 </tr>
25             </thead>
26
27             <!-- process each contact element -->
28             <xsl:for-each select = "contacts/contact">
29                 <tr>
30                     <td>
31                         <xsl:value-of select = "FirstName" />
32                     </td>
33                     <td>
34                         <xsl:value-of select = "LastName" />
35                     </td>
36                 </tr>
37             </xsl:for-each>
38
39         </table>
40
41     </body>
42
43 </html>
44
45 </xsl:template>
46
47 </xsl:stylesheet>

```

图 16.3 用于格式化联系人列表的 XSLT

脚本打印 HTTP 标头之后，第 15~18 行打开文件（如文件不能打开，就退出）。第 20 行打印根元素的 `<contacts>` 起始标记。第 23~27 行创建一个列表，该列表由 5 个元组构成。每个元组都包括两个值：一个字符及其对应的实体引用。第 30~43 行的 `for` 循环为文件中的每个姓名都生成 XML 元素。第 32~33 行调用 `replace` 方法将字符（例如 `<`、`&` 等等）替换成相应的实体引用。`split` 方法（第 36 行）从文件读入的一行中提取出姓氏和名字。第 37 行删除姓氏中的任何空白字符（比如回车）。包含姓名的 XML 元素在第 40~43 行进行打印。最后，第 47 行打印根元素的结束标记。

16.3 XML 处理包

本章后文将提供几个例子，借以说明如何使用 DOM（文档对象模型）和 SAX（Simple API for XML）进行 XML 处理的几个例子。本书写作时，Python 配套提供的 DOM 处理模块是 `xml.minidom` 和 `xml.pulldom`。注意，所有这些 DOM 实现都不完全相容于 W3C 的 DOM 推荐规范。所以，我们准备使用一个第三方的包，名为 4DOM。它完全相容于 W3C DOM 推荐规范。4DOM 是与 PyXML 包配套提供的（pyxml.sourceforge.net）。^①4DOM 提供的类和函数位于 `xml.dom.ext` 中。

16.5 节将使用与 Python[®] 配套提供的一个包（`xml.sax`），其中包含了进行 SAX 解析所需的类和函数。

在另一个名为 4XSLT 的包中，包含一个 XSLT 处理器，该处理器用于将 XML 文档转换成其他文本格式。4XSLT 位于 4Suite 这个包中（4suite.org）^②，它由 Fourthought 公司提供。4XSLT 提供的类和函数位于 `xml.xslt` 中。

^① 安装指南参见 www.deitel.com。

^② 需要 Python 2.0 或更高版本。

^③ PyXML 必须先于 4Suite 安装。安装指南参见 www.deitel.com。

16.4 文档对象模型 (DOM)

第15章介绍了文档对象模型 (DOM)。本节演示如何通过 Python 和 DOM API, 以程序化的方式来处理 XML 文档。

图 16.4 取得一个 XML 文档, 该文档标记了一篇文章 (图 16.5), 并用 4DOM 所实现的 DOM 来显示文档的元素名和值。

```

1 # Fig. 16.4: fig16_04.py
2 # Using 4DOM to traverse an XML Document.
3
4 import sys
5 from xml.dom.ext import StripXml
6 from xml.dom.ext.reader import PyExpat
7 from xml.parsers.expat import ExpatError
8
9 # open XML file
10 try:
11     file = open( "article2.xml" )
12 except IOError:
13     sys.exit( "Error opening file" )
14
15 # parse contents of XML file
16 try:
17     reader = PyExpat.Reader()          # create Reader instance
18     document = reader.fromStream( file ) # parse XML document
19     file.close()
20 except ExpatError:
21     sys.exit( "Error processing XML file" )
22
23 # get root element
24 rootElement = StripXml( document.documentElement )
25 print "Here is the root element of the document: %s" % \
26     rootElement.nodeName
27
28 # traverse all child nodes of root element
29 print "The following are its child elements:"
30
31 for node in rootElement.childNodes:
32     print node.nodeName
33
34 # get first child node of root element
35 child = rootElement.firstChild
36 print "\nThe first child of root element is:", child.nodeName
37 print "whose next sibling is:",
38
39 # get next sibling of first child
40 sibling = child.nextSibling
41 print sibling.nodeName
42 print 'Value of "%s" is:' % sibling.nodeName,
43
44 value = sibling.firstChild
45
46 # print text value of sibling
47 print value.nodeValue
48 print "Parent node of %s is: %s" % \
49     ( sibling.nodeName, sibling.parentNode.nodeName )
50
51 reader.releaseNode( document ) # remove DOM tree from memory

```

```

Here is the root element of the document: article
The following are its child elements:
title
date
author
summary
content

```

```

The first child of root element is: title
whose next sibling is: date
Value of "date" is: December 19, 2001
Parent node of date is: article

```

图 16.4 转换 XML 文档

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 16.5: article2.xml -->
4 <!-- Article formatted with XML -->
5
6 <article>
7
8   <title>Simple XML</title>
9
10  <date>December 19, 2001</date>
11
12  <author>
13    <firstName>Jane</firstName>
14    <lastName>Doe</lastName>
15  </author>
16
17  <summary>XML is easy.</summary>
18
19  <content>Once you have mastered XHTML, XML is learned
20    easily. Remember that XML is not for displaying
21    information but for managing information.
22  </content>
23
24 </article>

```

图 16.5 图 16.4 所用的 XML 文档

第 10~11 行尝试打开 article2.xml 以读取其中的内容。如果不能打开文件，程序会退出，并显示消息 "Error opening file"（第 12~13 行）。第 17 行实例化 PyExpat 的一个 Reader 对象，它是基于 DOM 的解析器的一个实例。PyExpat 模块位于 4DOM 的 reader 包中。第 18 行将 file 所引用的 XML 文档传给 Reader 的 fromStream 方法。该方法解析文档，并将 XML 文档的数据载入内存。document 变量引用 fromStream 所返回的 DOM 树（名为 Document）。

Document 对象的 documentElement 属性引用 Document 的根元素节点。第 24 行将根元素节点传给 4DOM 的 StripXml 函数。该函数从 XML DOM 树中删除多余的空白字符（例如回车换行符，或者用于缩进的空格）。如果不调用 StripXml，多余的空白字符会存储在 DOM 树中。第 15 章说过，DOM 树包含一系列节点。每个节点都是从 Node 类派生的一个类型，后文即将详细论述这些派生类。

第 25~26 行打印 rootElement 的名称，这是通过它的 nodeName 属性实现的。Node 对象的 childNodes 属性是一个列表，该列表由该节点的子构成。第 31~32 行打印 rootElement 的每个子节点的 nodeName。第 35~49 行打印特定节点的名称。Node 对象 firstChild 属性对应于子节点列表中的第一个子节点。第 35~36 行将 rootElement 的第一个子指派给变量 child，并打印这个“子”的名称。

第 40 行将 child 的下一个同辈节点指派给变量 sibling。nextSibling 属性包含节点的下一个同辈节点（也就是下一个具有相同父节点的节点）。例如，title，date，author，summary 和 content 都是同辈节点。第 41 行打印同辈节点的名称。

第 44 行将 sibling 的第一个子节点指派给变量 value。在这个例子中，value 是一个代表 sibling 内容的 Text 节点。Text 节点包含字符数据。第 47 行打印包含在 value 中的文本（通过访问它的 nodeValue 属性）。第 48~49 行打印 sibling 的父节点。父节点是通过 parentNode 属性取得的。最后，第 51 行调用 Reader 的 releaseNode 方法，从内存中清除指定的 Document（即 DOM 树）。

良好编程习惯 16.1 尽管在 Python 2.0 和更高版本中不需要调用 releaseNode 方法，但请坚持这样做，以确保从内存中清除 DOM 树。

从 Node 继承的类代表了不同的 XML 节点类型。Document 节点代表整个 XML 文档（在内存中），并提供了对其数据进行处理的方法；Element 节点代表 XML 元素；Text 节点代表字符数据；Attr 节点代表 XML 属性；而 Comment 节点代表注释。Document 节点可包含 Element、Text 和 Comment 节点。Element 节点可包含 Attr、Element、Text 和 Comment 节点。

图 16.6~图 16.12 总结了重要的 DOM 属性，以及用于浏览和更新 DOM 树的方法。图 16.6 描述了一些 Node 属性和方法；图 16.7 描述一些 NodeList（即包括各个 Node 的一个顺序列表）属性和方法；图 16.8 描述一些 NamedNodeMap（即包括各个 Node 的一个无序字典）属性和方法；图 16.9 描述一些 Document 属性方法；图 16.10 描述一些 Element 属性和方法；图 16.11 描述一些 Attr 属性；图 16.12 描述一个 Text 和 Comment 属性。

图 16.13 的程序使用 DOM 在联系人列表 XML 文档（即 contacts.xml，如图 16.14 所示）中添加姓名。XML 文档将载入内存，并在程序处理完之后，最后保存到磁盘（覆盖以前的版本）。

属性/方法	说明
<code>appendChild(newChild)</code>	将 <i>newChild</i> 追加到子节点列表。返回已追加的子节点
<code>attributes</code>	包含了当前节点的属性节点的 NamedNodeMap
<code>childNodes</code>	包含了节点当前的子节点的 NodeList
<code>firstChild</code>	NodeList 中的第一个子节点，或者为 None（如果当前节点没有子节点）
<code>insertBefore(newChild, refChild)</code>	将 <i>newChild</i> 节点插到 <i>refChild</i> 节点之前。 <i>refChild</i> 必须是当前节点的一个子节点；否则， <code>insertBefore</code> 会引发 <code>ValueError</code> 异常
<code>isSameNode(other)</code>	如果 <i>other</i> 是当前节点，就返回 <code>true</code>
<code>lastChild</code>	NodeList 中的最后一个子节点，或者为 None（如果当前节点没有子节点）
<code>nextSibling</code>	NodeList 中的下一个节点，或者为 None（如果节点没有下一个同辈节点）
<code>nodeName</code>	节点的名称，或者为 None（如果节点没有名称）
<code>nodeType</code>	代表节点类型的一个整数。Node 类定义了以下常量： ELEMENT_NODE = 1 ATTRIBUTE_NODE = 2 TEXT_NODE = 3 COMMENT_NODE = 8 DOCUMENT_NODE = 9
<code>nodeValue</code>	当前节点的值，或者为 None（如果节点没有值）
<code>parentNode</code>	父节点，或者为 None（如果节点没有父）
<code>previousSibling</code>	NodeList 中的上一个节点，或者为 None（如果节点没有上一个同辈节点）
<code>removeChild(oldChild)</code>	删除一个子点。 <i>oldChild</i> 必须是当前节点的一个子节点；否则会引发 <code>ValueError</code> 异常
<code>replaceChild(newChild, oldChild)</code>	将 <i>oldChild</i> 替换成 <i>newChild</i> 。 <i>oldChild</i> 必须是当前节点的一个子节点；否则 <code>replaceChild</code> 会引发一个 <code>ValueError</code> 异常

图 16.6 Node 属性和方法

属性/方法	说明
<code>item(i)</code>	返回索引 <i>i</i> 处的节点。索引范围从 0 到 <i>length</i> - 1
<code>length</code>	NodeList 中的节点个数

图 16.7 NodeList 属性和方法

属性/方法	说明
<code>item(i)</code>	返回索引 <i>i</i> 处的节点。索引范围从 0 到 <i>length</i> - 1
<code>length</code>	指定元素节点的属性节点个数

图 16.8 NamedNodeMap 属性方法

属性/方法	说明
<code>createAttribute(name)</code>	创建并返回具有指定 <i>name</i> (名称) 的一个 Attr 节点
<code>createComment(data)</code>	创建并返回包含指定 <i>data</i> (数据) 的一个 Comment 节点
<code>createElement(tagName)</code>	创建并返回具有指定 <i>tagName</i> 的一个 Element 节点
<code>createTextNode(data)</code>	创建并返回包含指定 <i>data</i> (数据) 的一个 Text 节点
<code>documentElement</code>	文档树 (DOM 树) 的根元素节点
<code>getElementsByTagName(name)</code>	返回子树中标记名为 <i>name</i> 的所有节点的一个 NodeList

图 16.9 Document 属性和方法

属性/方法	说明
<code>getAttribute(name)</code>	以字符串形式返回名称为 <i>name</i> 的 XML 属性的值
<code>getAttributeNode(name)</code>	返回名称为 <i>name</i> 的 XML 属性的 Attr 节点
<code>getElementsByTagName(name)</code>	返回子树中标记名为 <i>name</i> 的所有节点的一个 NodeList
<code>removeAttribute(name)</code>	在针对指定 Element 节点的 XML 属性列表中, 删除名为 <i>name</i> 的 XML 属性 (指定为一个字符串)
<code>removeAttributeNode(name)</code>	在针对指定 Element 节点的 XML 属性列表中, 删除名为 <i>name</i> 的 Attr 节点
<code>setAttribute(name, value)</code>	将名为 <i>name</i> 的 XML 属性的值修改成 <i>value</i> 。两个参数都指定字符串形式
<code>setAttributeNode(name)</code>	在针对指定 Element 节点的 XML 属性列表中, 添加名为 <i>name</i> 的新 Attr 节点。如属性已经存在, 新属性会替换当前属性
<code>tagName</code>	元素的标记名

图 16.10 Element 属性和方法

属性	说明
<code>name</code>	XML 属性名称。
<code>prefix</code>	命名空间前缀; 如果不存在, 就为 None

图 16.11 Attr 属性

属性	说明
<code>data</code>	节点 (Text 或 Comment) 的数据

图 16.12 Text 和 Comment 属性

```

1 # Fig. 16.13: fig16_13.py
2 # Using 4DOM to manipulate an XML Document.
3
4 import sys
5 from xml.dom.ext.reader import PyExpat
6 from xml.dom.ext import PrettyPrint
7
8 def printInstructions():
9     print """\nEnter 'a' to add a contact.
10 Enter 'l' to list contacts.xml.
11 Enter 'i' for instructions.
12 Enter 'q' to quit.""
13
14 def printList( document ):
15     print "Your contact list is:"
16
17     # iterate over NodeList of contact elements
18     for contact in document.getElementsByTagName( "contact" ):
19         first = contact.getElementsByTagName( "FirstName" )[ 0 ]
20
21         # get first node's value

```

```

22     firstText = first.firstChild.nodeValue
23
24     # get NodeList for nodes that contain tag name "LastName"
25     last = contact.getElementsByTagName( "LastName" )[ 0 ]
26     lastText = last.firstChild.nodeValue
27
28     print firstText, lastText
29
30 def addContact( document ):
31     root = document.documentElement # get root element node
32
33     name = raw_input(
34         "Enter the name of the person you wish to add: " )
35
36     first, last = name.split()
37
38     # create first name element node
39     firstNode = document.createElement( "FirstName" )
40     firstNodeText = document.createTextNode( first )
41     firstNode.appendChild( firstNodeText )
42
43     # create last name element node
44     lastNode = document.createElement( "LastName" )
45     lastNodeText = document.createTextNode( last )
46     lastNode.appendChild( lastNodeText )
47
48     # create contact node, append first name and last name nodes
49     contactNode = document.createElement( "contact" )
50     contactNode.appendChild( firstNode )
51     contactNode.appendChild( lastNode )
52
53     root.appendChild( contactNode ) # add contact node
54
55 # open contacts file
56 try:
57     file = open( "contacts.xml", "r+" )
58 except IOError:
59     sys.exit( "Error opening file" )
60
61 # create DOM parser and parse XML document
62 reader = PyExpat.Reader()
63 document = reader.fromStream( file )
64
65 printList( document )
66 printInstructions()
67 character = "l"
68
69 while character != "q":
70     character = raw_input( "\n? " )
71
72     if character == "a":
73         addContact( document )
74     elif character == "l":
75         printList( document )
76     elif character == "i":
77         printInstructions()
78     elif character != "q":
79         print "Invalid command!"
80
81 file.seek( 0, 0 ) # position to beginning of file
82 file.truncate() # remove data from file
83 PrettyPrint( document, file ) # print DOM contents to file
84 file.close() # close XML file
85 reader.releaseNode( document ) # free memory

```

```

Your contact list is:
John Black
Sue Green

Enter 'a' to add a contact.
Enter 'l' to list contacts.xml.

```

```

Enter 'i' for instructions.
Enter 'q' to quit.

? a
Enter the name of the person you wish to add: Michael Red

? l
Your contact list is:
John Black
Sue Green
Michael Red

? q

```

图 16.13 处理 XML 文档

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE contacts>
3 <contacts>
4   <contact>
5     <LastName>Black</LastName>
6     <FirstName>John</FirstName>
7   </contact>
8   <contact>
9     <LastName>Green</LastName>
10    <FirstName>Sue</FirstName>
11  </contact>
12  <contact>
13    <FirstName>Michael</FirstName>
14    <LastName>Red</LastName>
15  </contact>
16 </contacts>

```

图 16.14 图 16.13 输出的联系人列表

第 57 行打开 `contacts.xml` 以进行读写。第 62 行实例化一个解析器对象。第 63 行调用 `fromStream` 方法对 XML 文档进行解析并构建 DOM 树。

第 65 行调用 `printList` 函数（第 14~28 行），将联系人列表打印到屏幕。`getElementsByTagName` 函数（第 18 行）返回一个 `NodeList`，其中包含标记名为 `contact` 的所有 `Element` 节点。第 19 行调用 `getElementsByTagName` 函数以获得一个 `NodeList`，其中包含标记名为 `FirstName` 的所有 `Element` 节点。`contact` 引用的每个节点都只包含了一个这样的节点：这个节点作为列表中的第一个元素访问（即 `[0]`）。第 22 行将 `first` 的第一个子元素（一个 `Text` 节点）的值指派给变量 `firstText`。第 25~26 行重复这一过程以获得姓氏。第 28 行在屏幕上打印当前联系人的名字和姓氏。

第 66 行调用 `printInstructions` 函数，打印程序指示。第 69~79 行获得用户选择，并调用适当的函数。

`addContact` 函数（第 30~53 行）在列表中添加一个联系人。`Document` 的根元素是通过它的 `documentElement` 属性来获得的（第 31 行）。第 33~36 行提示用户输入，并调用字符串方法 `split` 将名字与姓氏分开。

第 39 行调用 `Document` 的 `createElement` 方法来创建一个标记名为 `FirstName`（名字）的 `Element` 节点。第 40~41 行创建一个 `Text` 节点，并把它追加到该 `Element` 节点——分别使用 `createTextNode` 和 `appendChild` 方法。第 44~46 行采用类似方式创建一个 `Element` 节点，它的标记名为 `LastName`（姓氏）。

第 49 行创建标记名为 `contact` 的一个 `Element` 节点。第 50~51 行调用 `appendChild` 方法，将 `firstNode` 和 `lastNode` 所引用的 `Element` 节点添加到 `contactNode` 所引用的节点。第 53 行调用 `appendChild` 方法，将 `contactNode` 引用的节点添加到 `root` 所引用的节点。

用户在联系人列表中完成添加姓名之后，选择保存文件。`seek` 方法（第 81 行）将文件指针定位到文件头，然后由 `truncate` 方法（第 82 行）删除文件内容。之后，4DOM 的 `PrettyPrint` 函数将更新的 XML 写入文件（第 83 行）。该函数可将 XML DOM 树的数据写入一个指定的输出流（并添加缩进和回车，以改善可读性）。第 84~85 行关闭文件，将 DOM 树从内存消除。

16.5 用 xml.sax 解析 XML

本节要讨论 `xml.sax` 包，它为基于 SAX 的解析提供了一系列模块。进行基于 SAX 的解析时，解析器会读取输入以识别 XML 标记。一旦解析器遇到标记，就会调用相应的事件处理程序（即方法）。例如，当解析器遇到一个起始标记时，会调用 `startElement` 事件处理程序；遇到字符数据时，会调用 `characters` 事件处理程序。程序员可覆盖事件处理程序，以便对 XML 进行特别的处理。图 16.15 总结了一些常用的 SAX 事件处理程序。

良好编程习惯 16.2 Python 联机文档提供了 `xml.sax` 事件处理程序的完整列表。网址是：
www.python.org/doc/current/lib/content-handler-objects.html。

事件处理程序	说明
<code>characters(content)</code>	解析器遇到字符数据时调用。字符数据 (<code>content</code>) 作为参数传给事件处理程序
<code>endDocument()</code>	解析器遇到文档尾时调用
<code>endElement(name)</code>	解析器遇到一个结束标记时调用。标记名 (<code>name</code>) 作为参数传给事件处理程序
<code>startDocument()</code>	解析器遇到文档头时调用
<code>startElement(name, attrs)</code>	解析器遇到一个起始标记时调用。标记名 (<code>name</code>) 及其属性 (<code>attrs</code>) 作为参数传给事件处理程序

图 16.15 `xml.sax` 事件处理程序（方法）

图 16.16 演示了基于 SAX 的解析。该程序允许用户指定将在 XML 文档中搜索的标记名。遇到标记名时，程序就输出元素的“属性-值”对。程序覆盖了 `startElement` 和 `endElement` 方法，处理遇到起始与结束标记时生成的事件。图 16.17 是该程序要使用的 XML 文档。

```

1 # Fig. 16.16: fig16_16.py
2 # Demonstrating SAX-based parsing.
3
4 from xml.sax import parse, SAXParseException, ContentHandler
5
6 class TagInfoHandler( ContentHandler ):
7     """Custom xml.sax.ContentHandler"""
8
9     def __init__( self, tagName ):
10         """Initialize ContentHandler and set tag to search for"""
11
12         ContentHandler.__init__( self )
13         self.tagName = tagName
14         self.depth = 0 # spaces to indent to show structure
15
16     # override startElement handler
17     def startElement( self, name, attributes ):
18         """An Element has started"""
19
20         # check if this is tag name for which we are searching
21         if name == self.tagName:
22             print "\n%s<%s> started" % ( " " * self.depth, name )
23
24             self.depth += 3
25
26             print "%sAttributes:" % ( " " * self.depth )
27
28             # check if element has attributes
29             for attribute in attributes.getNames():
30                 print "%s%s = %s" % ( " " * self.depth, attribute,
31                                     attributes.getValue( attribute ) )
32
33     # override endElement handler
34     def endElement( self, name ):

```

```

35     """An Element has ended"""
36
37     if name == self.tagName:
38         self.depth -= 3
39         print "%s</%s> ended\n" % ( " " * self.depth, name )
40
41 def main():
42     file = raw_input( "Enter a file to parse: " )
43     tagName = raw_input( "Enter tag to search for: " )
44
45     try:
46         parse( file, TagInfoHandler( tagName ) )
47
48     # handle exception if unable to open file
49     except IOError, message:
50         print "Error reading file:", message
51
52     # handle exception parsing file
53     except SAXParseException, message:
54         print "Error parsing file:", message
55
56 if __name__ == "__main__":
57     main()

```

```

Enter a file to parse: boxes.xml
Enter tag to search for: box

```

```

<box> started
  Attributes:
    size = big

  <box> started
    Attributes:
      size = medium
  </box> ended

  <box> started
    Attributes:
      type = small

    <box> started
      Attributes:
        type = tiny
    </box> ended

  </box> ended

</box> ended

```

图 16.16 基于 SAX 的解析示例

第 42~43 行获得要解析的 XML 文档名以及要查找的标记名。第 46 行调用 `xml.sax` 的 `parse` 函数，它创建一个 SAX 解析器对象。`parse` 函数的第一个参数可为 Python 文件对象或者文件名。但第二个参数必须是 `xml.sax.ContentHandler` 类的一个实例（或者是 `ContentHandler` 的一个派生类，比如 `TagInfoHandler`），它是 `xml.sax` 中的主要回调处理程序。在 `ContentHandler` 类中，包含了用于处理 SAX 事件的方法（参见图 16.15）。

打开指定文件时，如果出现错误，会引发一个 `IOError` 异常，第 50 行将显示一条错误消息。如果在解析文件期间出错（比如指定的 XML 文档不是良构的），`parse` 就会引发一个 `SAXParseException` 异常，第 54 行则会显示错误消息。

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 16.17: boxes.xml -->
4 <!-- XML document used in Fig. 16.16 -->
5
6 <boxlist>

```

```

7
8 <box size = "big">
9   This is the big box.
10
11   <box size = "medium">
12     Medium sized box
13     <item>Some stuff</item>
14     <thing>More stuff</thing>
15   </box>
16
17   <parcel />
18   <box type = "small">
19     smaller stuff
20     <box type = "tiny">tiny stuff</box>
21   </box>
22
23 </box>
24
25 </boxlist>

```

图 16.17 在图 16.16 中使用的 XML 文档

我们的例子只覆盖了两个事件处理程序。遇到开始标记和结束标记时，会分别调用 `startElement` 和 `endElement` 方法。其中，`startElement` 方法（第 16~31 行）要取得两个参数：元素的标记名（字符串形式）以及元素的属性。属性作为 `AttributeImpl` 类的一个实例传递给方法，这个类是在 `xml.sax.reader` 中定义的。该类为元素的属性提供了一个字典风格的接口。

第 21 行判断从事件中收到的元素是否包含用户指定的标记名。如果是，第 22 行就打印起始标记，并缩进指定空距（由 `depth` 指定），而且第 24 行使 `depth` 自增 3，确保打印的下一个标记进一步缩进。

第 29~31 行打印元素的属性。`for` 循环首先调用 `attributes` 的 `getNames` 方法，以获得属性名。然后，循环打印每个属性名及其相应的值（将当前属性名传给 `attributes` 的 `getValue` 方法，即可获得值）。

遇到结束标记时，会执行 `endElement` 方法（第 34~39 行），并取得结束标记的名称作为参数。如果 `name` 包含由用户指定的标记名，第 38 行使 `depth` 自减，从而减少缩进量。第 39 行打印找到的结束标记。

16.6 案例分析：用 Python 和 XML 实现论坛^①

本节利用 XML 以及几种相关技术创建一种流行的网站应用：论坛程序（Message Forum）。它是一种虚拟的公告牌，用户可在上面讨论各种主题。论坛常见的功能包括讨论组、问题与解答区域以及意见反馈区。许多网站都支持论坛。一些流行的论坛包括：

groups.yahoo.com
web.eesite.com/forums
groups.google.com

图 16.18 总结了构成论坛的文件。图 16.19 显示了所需的目录结构，它适用于 Windows 上运行的 Apache 服务器。图 16.20 展示了文件之间的一些关键交互。`default.py` 生成的 XHTML 主页显示了可用论坛的一个列表，它们存储在 XML 文档 `forums.xml` 中。这个 XHTML 主页还为每个 XML 论坛文档提供了超链接以及到脚本程序 `addForum.py` 的超链接，它负责为 `forums.xml` 添加一个论坛，并以 `template.xml` 中的标记内容作为起点，实际创建一个 XML 论坛。

文件名	说明
<code>forums.xml</code>	包含可用论坛标题及其文件名的 XML 文档
<code>default.py</code>	论坛主页，提供到各个论坛的导航链接

^① 我们实现的论坛需要 Internet Explorer 5.0 以及 msxml 3.0（或者它们的更高版本）。16.6.3 节将讨论如何使用其他客户端浏览器，比如 Netscape。

文件名	说明
template.xml	论坛文档所用的模板
addForum.py	用于添加新论坛的脚本程序
feedback.xml	示范论坛
formatting.xsl	用于将论坛转换成 XHTML 的 XSLT 文档
addPost.py	为论坛添加文章的脚本程序
error.html	显示一条错误消息
site.css	用于格式化 XHTML 内容的样式表
forum.py	可在服务器上 将 XML 文档转换成 HTML 的脚本程序，目的是支持非 Internet Explorer 的客户端浏览器

图 16.18 论坛文档

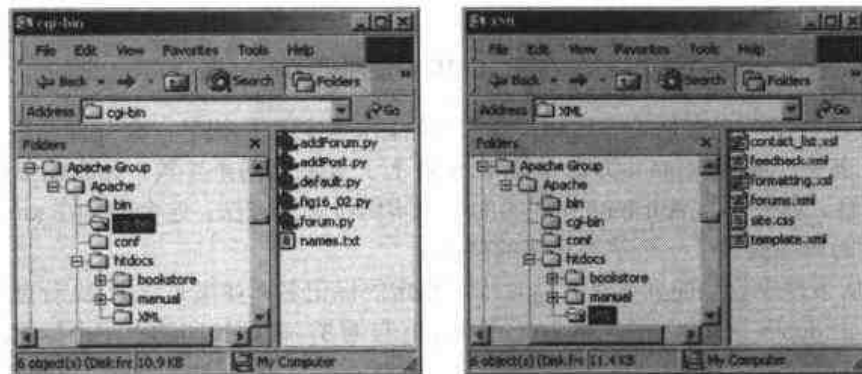


图 16.19 论坛的目录结构

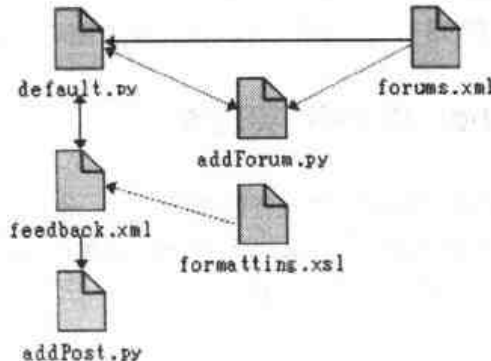


图 16.20 论坛各文档之间的关键交互

16.6.1 显示论坛

本节说明如何用 XML 标记论坛数据，并解释用于创建 XHTML 论坛主页的 Python 脚本程序 default.py。针对这个案例分析，我们提供了一个示范论坛（图 16.21），名为 feedback.xml，以便展示论坛文档的结构。

```
1 <?xml version = "1.0"?>
2
3 <!-- Fig. 16.2.1: feedback.xml -->
4 <!-- XML document representing a forum -->
5
6 <?xml:stylesheet type = "text/xsl" href = "../XML/formatting.xsl"?>
7
8 <forum file = "feedback.xml">
9   <name>Feedback</name>
```



```

10
11 <message timestamp = "Wed Jun 27 12:53:22 2001">
12   <user>Jessica</user>
13   <title>Nice forums!</title>
14   <text>These forums are great! Well done, all.</text>
15 </message>
16
17 </forum>

```

图 16.21 用于表示一个论坛的 XML 文档（内含一篇文章）

第 6 行引用了样式表 `formatting.xml`。如果由 `msxml` 进行应用，该 XSLT 文档（本章稍后将讨论）会将 XML 转换成 XHTML，以便在 Internet Explorer 上显示。论坛文档具有根元素 `forum`，其中含有 `file` 属性。该属性的值就是文档的文件名。子元素包括用于指定论坛标题的 `name`，以及用于标记文章的 `message`。在文章中包含用户名、标题和正文，分别由 `user`、`title` 和 `text` 进行标记。另外，还会为文章指定一个 `timestamp`（时间标记）。

`forums.xml` 文档（图 16.22）包含对应每个论坛的文件名和标题。论坛创建时，这个文档就会更新。

在图 16.22 中，根元素 `forums`（第 8 行）包含一个或多个 `forum` 子元素。最初只存在一个论坛（即示范性的 Feedback）。每个 `forum` 元素都有 `filename` 属性以及子元素 `name`。

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 16.22: forums.xml          -->
4 <!-- XML document containing all forums -->
5
6 <?xml:stylesheet type = "text/xsl" href = "formatting.xml"?>
7
8 <forums>
9
10   <forum filename = "feedback.xml">
11     <name>Feedback</name>
12   </forum>
13
14 </forums>

```

图 16.22 列出了所有可用论坛的 XML 文档

论坛访问者最开始看到的是由 `default.py`（图 16.23）生成的网页，其中列出了到所有论坛的链接，并提供了论坛管理选项。最初只有两个活动链接，一个用于查看 Feedback 论坛（示范论坛），另一个用于创建论坛。

这个 Python 脚本使用 4DOM 包中的模块来解析 `forums.xml`。第 33~34 行实例化一个解析器对象，然后载入并解析 `forums.xml`。第 38~71 行将 XHTML 输出到浏览器。首先，第 38 行调用 `printHeader` 函数（第 9~23 行），打印主页的 XHTML 头。该函数在打印 XHTML 头时，使用一个指定的标题和一个到 CSS 文件的链接（用于格式化网页）。在这个案例分析中，我们要利用 `msxml` 的 XML 解析及 XSLT 处理能力来减少服务器的工作量。第 44~45 行判断客户是否使用 Internet Explorer。如果是，就将 `prefix`（前缀）设为 `"/XML/"`。否则将 `prefix` 设为 `"forum.py?file="`。注意，第 47 行使用 `prefix` 构建到每个论坛的超链接。使用 Internet Explorer 的客户直接请求 XML 文档，其他客户则请求 `forum.py`。16.6.3 节将进一步讨论这方面的问题。`for` 循环（第 50~60 行）获取包含标记名 `forum` 的所有 `Element` 节点。针对 `forums.xml` 中发现的每个论坛，都会创建超链接。第 62~71 行打印剩余的 XHTML，其中包括到 `addForum.py` 脚本的一个链接。最后，第 73 行将 `Document` 对象从内存清除。

```

1 #!c:\Python\python.exe
2 # Fig. 16.23: default.py
3 # Default page for message forums.
4
5 import os
6 import sys
7 from xml.dom.ext.reader import PyExpat
8

```

```

9 def printHeader( title, style ):
10     print """Content-type: text/html
11
12 <?xml version = "1.0" encoding = "UTF-8"?>
13 <!DOCTYPE html PUBLIC
14     "-//W3C//DTD XHTML 1.0 Strict//EN"
15     "DTD/xhtml1-strict.dtd">
16 <html xmlns = "http://www.w3.org/1999/xhtml">
17
18 <head>
19 <title>%s</title>
20 <link rel = "stylesheet" href = "%s" type = "text/css" />
21 </head>
22
23 <body>""" % ( title, style )
24
25 # open XML document that contains the forum names and locations
26 try:
27     XMLFile = open( "../htdocs/XML/forums.xml" )
28 except IOError:
29     print "Location: /error.html\n"
30     sys.exit()
31
32 # parse XML document containing forum information
33 reader = PyExpat.Reader()
34 document = reader.fromStream( XMLFile )
35 XMLFile.close()
36
37 # write XHTML to browser
38 printHeader( "Deitel Message Forums", "/XML/site.css" )
39 print """<h1>Deitel Message Forums</h1>
40 <p style="font-weight:bold">Available Forums</p>
41 <ul>"""
42
43 # determine client-browser type
44 if os.environ[ "HTTP_USER_AGENT" ].find( "MSIE" ) != -1:
45     prefix = "../XML/" # Internet Explorer
46 else:
47     prefix = "forum.py?file="
48
49 # add links for each forum
50 for forum in document.getElementsByTagName( "forum" ):
51
52     # create link to forum
53     link = prefix + forum.attributes.item( 0 ).value
54
55     # get element nodes containing tag name "name"
56     name = forum.getElementsByTagName( "name" )[ 0 ]
57
58     # get Text node's value
59     nameText = name.childNodes[ 0 ].nodeValue
60     print '<li><a href = "%s">%s</a></li>' % ( link, nameText )
61
62 print """</ul>
63 <p style="font-weight:bold">Forum Management</p>
64 <ul>
65     <li><a href = "addForum.py">Add a Forum</a></li>
66     <li>Delete a Forum</li>
67     <li>Modify a Forum</li>
68 </ul>
69 </body>
70
71 </html>"""
72
73 reader.releaseNode( document )

```

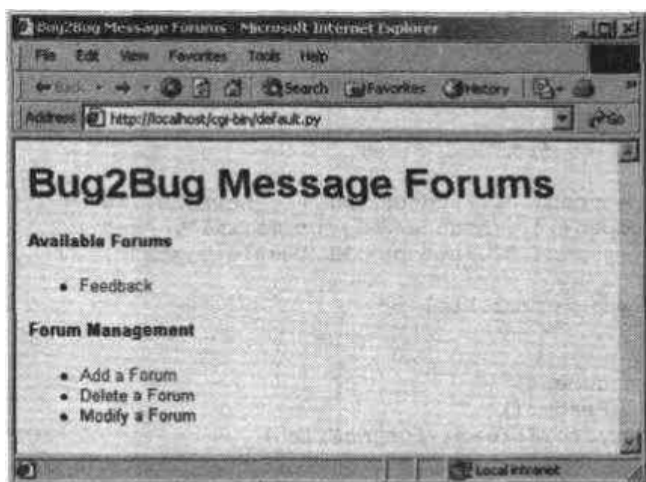


图 16.23 论坛的默认页

16.6.2 添加论坛和文章

本节讨论负责添加论坛和文章的 Python 脚本和文档。图 16.24 是用于添加新论坛的 Python 脚本。这个脚本使用 4DOM 中的模块处理 XML 文档。

最开始请求脚本时，不传递任何参数。脚本首先获取表单数据（第 29 行）。由于表单不包含任何值，所以先执行第 93 行的 else 块。第 94~107 行输出一个表单，提示用户输入论坛名称，以及要创建的 XML 文档的文件名。表单提交之后，会重新请求执行脚本，并向其传递用户在表单中输入的值。此时，第 32 行的条件为 true，所以要执行第 33~92 行。

```

1  #!c:\Python\python.exe
2  # Fig. 16.24: addForum.py
3  # Adds a forum to the list
4
5  import re
6  import sys
7  import cgi
8
9  # 4DOM packages
10 from xml.dom.ext.reader import PyExpat
11 from xml.dom.ext import PrettyPrint
12
13 def printHeader( title, style ):
14     print """Content-type: text/html
15
16 <?xml version = "1.0" encoding = "UTF-8"?>
17 <!DOCTYPE html PUBLIC
18     "-//W3C//DTD XHTML 1.0 Strict//EN"
19     "DTD/xhtml1-strict.dtd">
20 <html xmlns = "http://www.w3.org/1999/xhtml">
21
22 <head>
23 <title>%s</title>
24 <link rel = "stylesheet" href = "%s" type = "text/css" />
25 </head>
26
27 <body>""" % ( title, style )
28
29 form = cgi.FieldStorage()
30
31 # if user enters data in form fields
32 if form.has_key( "name" ) and form.has_key( "filename" ):
33     newFile = form[ "filename" ].value
34
35     # determine whether file has xml extension

```

```

36 if not re.match( "\w+\.xml$", newFile ):
37     print "Location: /error.html\n"
38     sys.exit()
39 else:
40
41     # create forum files from xml files
42     try:
43         newForumFile = open( "../htdocs/XML/" + newFile, "w" )
44         forumsFile = open( "../htdocs/XML/forums.xml", "r+" )
45         templateFile = open( "../htdocs/XML/template.xml" )
46     except IOError:
47         print "Location: /error.html\n"
48         sys.exit()
49
50     # parse forums document
51     reader = PyExpat.Reader()
52     document = reader.fromStream( forumsFile )
53
54     # add new forum element
55     forum = document.createElement( "forum" )
56     forum.setAttribute( "filename", newFile )
57
58     name = document.createElement( "name" )
59     nameText = document.createTextNode( form[ "name" ].value )
60     name.appendChild( nameText )
61     forum.appendChild( name )
62
63     # obtain root element of forum
64     documentNode = document.documentElement
65     firstForum = documentNode.getElementsByTagName(
66         "forum" )[ 0 ]
67     documentNode.insertBefore( forum, firstForum )
68
69     # write updated XML to disk
70     forumsFile.seek( 0, 0 )
71     forumsFile.truncate()
72     PrettyPrint( document, forumsFile )
73     forumsFile.close()
74
75     # create document for new forum from template file
76     document = reader.fromStream( templateFile )
77     forum = document.documentElement
78     forum.setAttribute( "file", newFile )
79
80     # create name element
81     name = document.createElement( "name" )
82     nameText = document.createTextNode( form[ "name" ].value )
83     name.appendChild( nameText )
84     forum.appendChild( name )
85
86     # write generated XML to new forum file
87     PrettyPrint( document, newForumFile )
88     newForumFile.close()
89     templateFile.close()
90     reader.releaseNode( document )
91
92     print "Location: default.py\n"
93 else:
94     printHeader( "Add a forum", "/XML/site.css" )
95     print "====<form action = \"addForum.py\" method=\"post\">
96 Forum Name<br />
97 <input type = \"text\" name = \"name\" size = \"40\" /><br />
98 Forum File Name<br />
99 <input type = \"text\" name = \"filename\" size = \"40\" /><br />
100 <input type = \"submit\" name = \"submit\" value = \"Submit\" />
101 <input type = \"reset\" value = \"Reset\" />
102 </form>
103
104 <a href = \"/cgi-bin/default.py\">Return to Main Page</a>
105 </body>
106

```

```
107 </html>"""
```

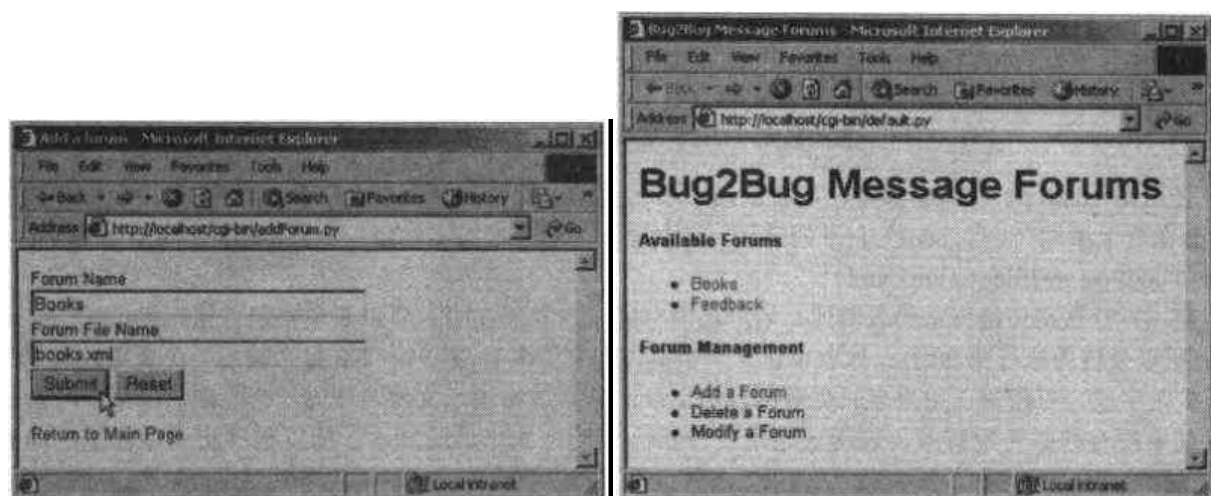


图 16.24 向 forums.xml 添加新论坛的脚本

第 36 行检查传给脚本的文件名，确定其中只包括字母或数字，而且以.xml 结尾；否则，脚本会将客户浏览器重定向至 error.html。这样可防止恶意用户向一个系统文件写入数据，或者骗取对服务器的无限访问途径（注意，这些问题还可采用其他解决方案，比如在服务器上生成文件名）。如文件名合法，第 43 行就调用 open 函数，尝试创建文件。

第 44 行打开 forums.xml 文件以便进行读写（"r+"）。第 40 行打开名为 template.xml 的模板 XML 文档（如图 16.25 所示），它提供了一个论坛的基本结构。模板中包含一个空的 forums 元素，将以程序化的方式为其添加论坛名称和文件名。打开任何文件时如果出现错误，就把客户浏览器重定向到 error.html。

第 51 行实例化一个 DOM 解析器，并把它指派给变量 reader。第 52 行载入并解析 forums.xml；创建的 Document 对象指派给变量 document。由于我们希望在 forums 内部创建一个 forum 元素，所以第 55 行调用 Document 对象的 createElement 方法，并指定新元素的名称（"forum"）。新 Element 节点的 filename 属性通过调用 setAttribute 方法进行设置（同时传递属性的名称和值）。

forum 元素只包含一种信息，即第 58~61 行所添加的论坛名称。第 58 行创建另一个 Element 节点，名为 name。为了向新的 Element 节点添加字符数据，必须创建一个子 Text 节点。所以，我们调用 createTextNode 方法（第 59 行），并传递来自表单的论坛名称（即 form["name"].value）。第 60 行调用 appendChild 方法，将 Text 节点追加到 name 所引用的 Element 节点。第 61 行将 name 引用的 Element 节点添加到由 forum 引用的 Element 节点。

第 64 行访问 document 的 documentElement 属性，以获取根元素节点（即 forums）。第 65~66 行调用 getElementsByTagName 方法，获得由所有 forum 元素构成的一个 NodeList，并将第一个元素指派给变量 firstForum。第 67 行调用 insertBefore 方法，将 forum 引用的新 Element 节点插到 forums 的第一个子节点之前。这样一来，最近添加的论坛会先在论坛列表中出现。

为更新 forums.xml，第 70 行使用 seek 方法定位到文件头，并删除现有的全部数据（将文件长度截短为 0）。然后，第 72 行调用 PrettifyPrint 函数，将更新好的 XML 写入 forumsFile。

第 76 行调用 fromStream 方法，并将创建的 Document 对象指派给变量 document，从而载入并解析 template.xml 文件（图 16.25）。第 77 行使用 documentElement 获得根元素，第 78 行把它的 file 属性值设为指定的文件名。第 81~84 行添加 name 节点，而第 87~88 行将更新的 XML 输出到 newForumFile，并关闭文件。第 89~90 行关闭 template.xml，并从内存中清除 Document 对象。随后，第 92 行将用户重定向到 default.py。

```
1 <?xml version = "1.0"?>
2
3 <!-- Fig. 16.25: template.xml -->
```

```

4 <!-- Empty forum file      -->
5
6 <?xml:stylesheet type = "text/xsl" href = "../XML/formatting.xsl"?>
7 <forum>
8 </forum>

```

图 16.25 用于生成新论坛的 XML 模板

使用图 16.26 的 Python 脚本，可在论坛中添加文章。将 formatting.xsl（图 16.27）应用于论坛文档后，会在页上添加一个到 addPost.py 的链接，其中包含当前论坛的文件名。这个文件名会传给 addPost.py（例如 addPost.py?file=forum1.xml）。

第 37 行获得传给脚本的表单值。用户还没有提交一篇新文章，所以表单不包含值“submit”（第 40 行），所以会转为执行第 90 行。如表单包含一个值（即文件名），第 91~108 行会输出一个表单，其中包含了多个字段，以便输入用户名、文章标题和文章正文，并将论坛文件名作为一个隐藏值（第 99 行）。注意如果没有参数传给脚本，表明这是用一种不恰当的方式来访问脚本，所以程序将浏览器重定向到 error.html（第 110 行）。

提交了表单数据后，会对提交的信息进行处理（从第 41 行开始）。如图所示，第 44~52 行会检查文件名中是否含有.xml 扩展名，并打开文件。第 55~61 行解析论坛文件，创建一个具有标记名 message 的 Element 节点。然后，调用 setAttribute 方法，设置节点的 timestamp 属性。

```

1  #!c:\Python\python.exe
2  # Fig. 16.26: addPost.py
3  # Adds a message to a forum.
4
5  import re
6  import os
7  import sys
8  import cgi
9  import time
10
11  # 4DOM packages
12  from xml.dom.ext.reader import PyExpat
13  from xml.dom.ext import PrettyPrint
14
15  def printHeader( title, style ):
16      print """Content-type: text/html
17
18  <?xml version = "1.0" encoding = "UTF-8"?>
19  <!DOCTYPE html PUBLIC
20      "-//W3C//DTD XHTML 1.0 Strict//EN"
21      "DTD/xhtml1-strict.dtd">
22  <html xmlns = "http://www.w3.org/1999/xhtml">
23
24  <head>
25  <title>%s</title>
26  <link rel = "stylesheet" href = "%s" type = "text/css" />
27  </head>
28
29  <body>""" % ( title, style )
30
31  # identify client browser
32  if os.environ[ "HTTP_USER_AGENT" ].find( "MSIE" ) != -1:
33      prefix = "../XML/" # Internet Explorer
34  else:
35      prefix = "forum.py?file="
36
37  form = cgi.FieldStorage()
38
39  # user has submitted message to post
40  if form.has_key( "submit" ):
41      filename = form[ "file" ].value
42
43      # add message to forum
44      if not re.match( "\w+\.xml$", filename ):

```

```

45     print "Location: /error.html\n"
46     sys.exit()
47
48     try:
49         forumFile = open( "../htdocs/XML/" + filename, "r+" )
50     except IOError:
51         print "Location: /error.html\n"
52         sys.exit()
53
54     # parse forum document
55     reader = PyExpat.Reader()
56     document = reader.fromStream( forumFile )
57     documentNode = document.documentElement
58
59     # create message element
60     message = document.createElement( "message" )
61     message.setAttribute( "timestamp", time.ctime( time.time() ) )
62
63     # add elements to message
64     messageElements = [ "user", "title", "text" ]
65
66     for item in messageElements:
67
68         if not form.has_key( item ):
69             text = "( Field left blank )"
70         else:
71             text = form[ item ].value
72
73         # create nodes
74         element = document.createElement( item )
75         elementText = document.createTextNode( text )
76         element.appendChild( elementText )
77         message.appendChild( element )
78
79     # append new message to forum and update document on disk
80     documentNode.appendChild( message )
81     forumFile.seek( 0, 0 )
82     forumFile.truncate()
83     PrettyPrint( document, forumFile )
84     forumFile.close()
85     reader.releaseNode( document )
86
87     print "Location: %s\n" % ( prefix + form[ "file" ].value )
88
89     # create form to obtain new message
90     elif form.has_key( "file" ):
91         printHeader( "Add a posting", "/XML/site.css" )
92         print ""\n<form action = "addPost.py" method="post">
93         User<br />
94         <input type = "text" name = "user" size = "40" /><br />
95         Message Title<br />
96         <input type = "text" name = "title" size = "40" /><br />
97         Message Text<br />
98         <textarea name = "text" cols = "40" rows = "5"></textarea><br />
99         <input type = "hidden" name = "file" value = "%s" />
100        <input type = "submit" name = "submit" value = "Submit" />
101        <input type = "reset" value = "Reset" />
102        </form>
103
104        <a href = "%s">Return to Forum</a>
105        </body>
106
107        </html>"" % ( form[ "file" ].value,
108            prefix + form[ "file" ].value )
109    else:
110        print "Location: /error.html\n"

```

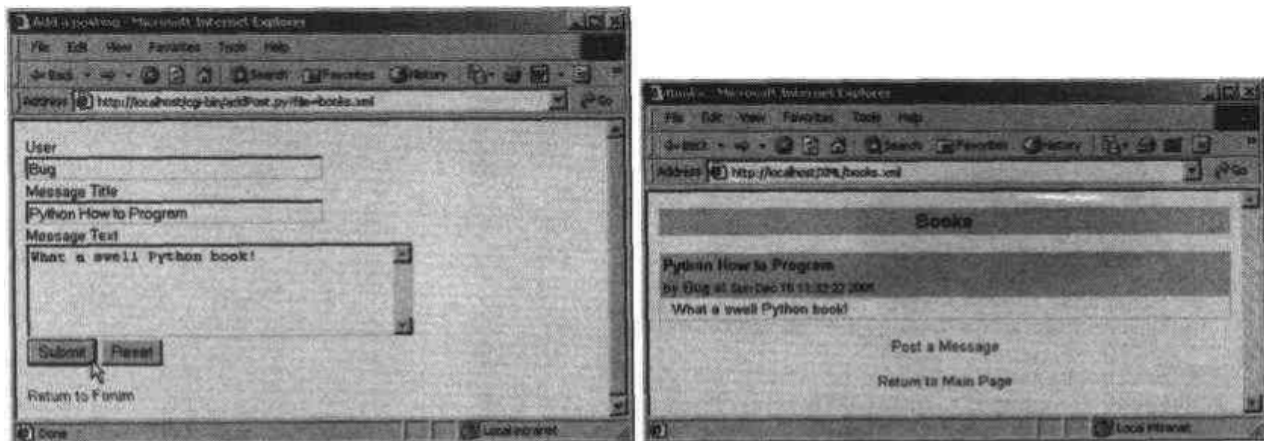


图 16.26 为论坛添加文章的脚本

第 64~77 行创建了分别代表 user, title 和 text 的 Element 节点, 并添加与表单中输入的值对应的文本。要注意的是, 如果某个字段留空未填, 会为那个字段输入“(Field left blank)”。每个新的 Element 节点都追加到 message 所引用的节点中(第 77 行)。

第 80 行将 message 引用的节点追加到 documentNode 所引用的节点。第 81~82 行对 XML 文件进行 seek 和 truncate 操作, 删除文件内容, 再写入更新的 XML 内容。第 84~85 行关闭文件, 并从内存清除 Document 对象。最后, 第 87 行将用户重定向到更新过的 XML 文档。

16.6.3 照顾不支持 XML 和 XSLT 的浏览器

这个案例分析用一个 XSLT 样式表(如图 16.27 所示的 formatting.xsl)将 XML 数据转换成 XHTML, 以便在 Internet Explorer 中呈现数据。记住, 发送给 Internet Explorer 的每个 XML 文档都包含一个引用了该样式表的处理指令。

```

1  <?xml version = "1.0"?>
2
3  <!-- Fig. 16.27: formatting.xsl -->
4  <!-- Style sheet for forum files -->
5
6  <xsl:stylesheet version = "1.0"
7    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
8
9    <!-- match document root -->
10   <xsl:template match = "/">
11     <html xmlns = "http://www.w3.org/1999/xhtml">
12
13       <!-- apply templates for all elements -->
14       <xsl:apply-templates select = "*" />
15     </html>
16   </xsl:template>
17
18   <!-- match forum elements -->
19   <xsl:template match = "forum">
20     <head>
21       <title><xsl:value-of select = "name" /></title>
22       <link rel = "stylesheet" type = "text/css"
23         href = "../XML/site.css" />
24     </head>
25
26     <body>
27       <table width = "100%" cellpadding = "0"
28         cellspacing = "2">
29         <tr>
30           <td class = "forumTitle">

```



```

31         <xsl:value-of select = "name" />
32     </td>
33 </tr>
34 </table>
35
36 <!-- apply templates for message elements -->
37 <br />
38 <xsl:apply-templates select = "message" />
39 <br />
40
41 <div style = "text-align: center">
42     <a>
43
44         <!-- add href attribute to "a" element -->
45         <xsl:attribute name = "href">../cgi-bin/addPost.py?file=<xsl:value-of select
= "@file" />
46         </xsl:attribute>
47         Post a Message
48     </a>
49     <br /><br />
50     <a href = "../cgi-bin/default.py">Return to Main Page</a>
51 </div>
52
53 </body>
54 </xsl:template>
55
56 <!-- match message elements -->
57 <xsl:template match = "message">
58     <table width = "100%" cellspacing = "0"
59         cellpadding = "2">
60         <tr>
61             <td class = "msgTitle">
62                 <xsl:value-of select = "title" />
63             </td>
64         </tr>
65
66         <tr>
67             <td class = "msgInfo">
68                 by
69                 <xsl:value-of select = "user" />
70                 at
71                 <span class = "date">
72                     <xsl:value-of select = "@timestamp" />
73                 </span>
74             </td>
75         </tr>
76
77         <tr>
78             <td class = "msgText">
79                 <xsl:value-of select = "text" />
80             </td>
81         </tr>
82     </table>
83 </xsl:template>
84 </xsl:stylesheet>
85
86 </xsl:stylesheet>

```

图 16.27 将 XML 转换成 XHTML 的 XSLT 样式表

目前只有 Internet Explorer 5 和更高版本才支持 XSLT。这意味着我们的论坛应用程序不能将 XML 内容发送给没有内建 XML 解析器和 XSLT 处理器的浏览器，比如 Netscape Communicator 6。为了创建不依赖于客户端浏览器的应用程序，可在服务器上解析 XML，并在服务器上应用 XSLT 转换。

在 4Suite 包括的 4XSLT 包中，提供了一个可将 XML 转换成 HTML 的 XSLT 处理器。可创建该处理器的一个实例，以便将样式表应用于 XML 文档。

在 default.py（图 16.23）和 addPost.py（图 16.26）中，我们用 prefix 变量指定要将客户端浏览器链

接或重定向到什么地方。由 Internet Explorer 的 XML 解析器和 XSLT 处理器来解析 XML 并为其应用一个样式表，可减轻服务器的负担。但是，对于不支持 XML 和 XSLT 的浏览器，需要将客户定向到一个特殊的 Python 脚本，以便在服务器上解析 XML 文档，并将 HTML 发送给客户。

所以，我们在 default.py 的第 44 行以及 addPost.py 的第 32 行插入浏览器检测代码：

```
if os.environ[ "HTTP_USER_AGENT" ].find( "MSIE" ) != -1:
    prefix = "../XML/"
else:
    prefix = "forum.py?file="
```

变量 prefix 会根据 HTTP_USER_AGENT 环境变量中是否出现 MSIE（即 Microsoft Internet Explorer）而进行设置。为简化问题，我们假定 Internet Explorer 5 或更高版本（配合 msxml 3.0 或更高版本）是唯一存在的 MSIE 版本，不检测更老的 MSIE 版本。

设置好 prefix 后，我们用它的值自定义脚本生成的 URL。例如 addPost.py 的第 87 行：

```
print "Location: %s\n" % ( prefix + form[ "file" ].value )
```

这一行将 Internet Explorer 用户定向到指定的 XML 论坛文件，该文件位于 ../XML/ 目录中，其他浏览器的用户则被定向到 forum.py。后者是一个 Python 脚本，只接收一个参数（即文件名）。

图 16.28 是 forum.py 的源代码，它在服务器上上 XML 文档转换成 HTML。该图还展示了 Netscape Communicator 的示范 HTML 输出。

```
1  #!c:\Python\python.exe
2  # Fig. 16.28: forum.py
3  # Display forum postings for non-Internet Explorer browsers.
4
5  import re
6  import cgi
7  import sys
8  from xml.xslt import Processor
9
10 form = cgi.FieldStorage()
11
12 # form to display has been specified
13 if form.has_key( "file" ):
14
15     # determine whether file is xml
16     if not re.match( "\w+\.xml$", form[ "file" ].value ):
17         print "Location: /error.html\n"
18         sys.exit()
19
20     try:
21         style = open( "../htdocs/XML/formatting.xml" )
22         XMLFile = open( "../htdocs/XML/" + form[ "file" ].value )
23     except IOError:
24         print "Location: /error.html\n"
25         sys.exit()
26
27     # create XSLT processor instance
28     processor = Processor.Processor()
29
30     # specify style sheet
31     processor.appendStylesheetStream( style )
32
33     # apply style sheet to XML document
34     results = processor.runStream( XMLFile )
35     style.close()
36     XMLFile.close()
37     print "Content-type: text/html\n"
38     print results
39 else:
40     print "Location: /error.html\n"
```

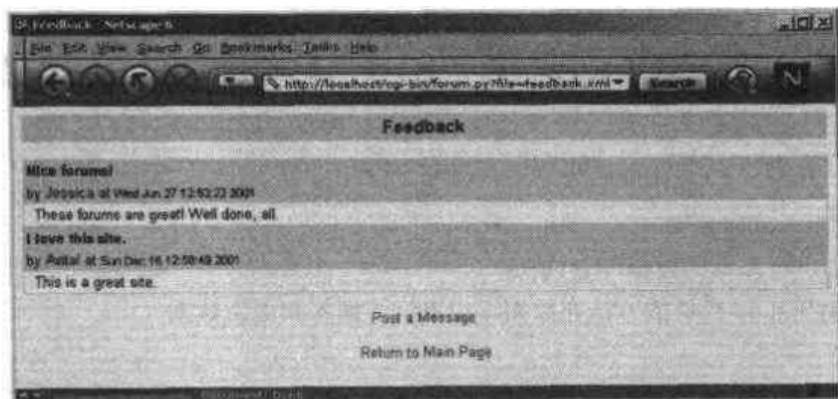


图 16.28 为不支持 XSLT 的浏览器将 XML 转换成 HTML

如果没有向脚本传递一个文件名, 用户会被重定向到 `error.html` (第 40 行), 否则就从第 16 行开始执行。第 16~18 行判断指定的文件名是否以 `.xml` 结尾。如果是, 第 21~22 行会分别打开 XSLT 样式表 `formatting.xsl` 以及指定的 XML 文档。如打开文件时出错, 用户会被重定向到 `error.html` (第 24 行)。

然后将 XML 转换成 HTML 以便显示。第 28 行实例化一个 `4XSLT Processor` (处理器) 对象, 它通过应用一个 XSLT 样式表将 XML 转换成 HTML。第 31 行通过调用 `processor` 的 `appendStyleSheetStream` 方法, 从而指定恰当的 XSLT 样式表。在可由 `Processor` 使用的样式表的列表中, 该方法追加一个样式表。注意, 可追加多个样式表 (也就是说, 可多次调用 `appendStyleSheetStream`), 以便利用同一个 `Processor` 对象, 将 XML 文档转换成许多不同的格式。传给 `appendStyleSheetStream` 的参数必须是一个 Python 文件对象。可为 `4XSLT Processor` 追加样式表的其他方法包括 `appendStyleSheetString`, `appendStyleSheetNode` 和 `appendStyleSheetUri`, 它们的参数分别是包含 XSLT 样式表的一个字符串、包含样式表的一个 DOM 树以及引用样式表的一个 URI。指定的 URI 可以是一个 URL (采用字符串形式), 以指定 Web 上一个样式表的位置。第 34 行调用 `Processor` 的 `runStream` 方法, 将样式表应用于 XML 文档。和 `appendStyleSheetStream` 一样, 传给 `runStream` 的对象必须是一个 Python 文件对象。用于应用样式表的其他方法包括 `runString`, `runNode` 和 `runUri`, 它们的参数分别为包含 XML 的一个字符串、包含 XML 的一个 DOM 树以及引用 XML 文档的一个 URI。

第 35~36 行关闭脚本所用的 XSLT 文件和 XML 文件。第 37 行为 Web 浏览器打印内容类型头。然后, 转换过的 XML 以 HTML 形式发送给客户 (第 38 行)。

本章利用第 15 章介绍的概念创建基于 XML 的应用程序。我们利用包含 DOM 和 SAX 实现的 Python 包来解析 XML 文档, 然后用 XSLT 样式表在浏览器中显示 XML 文档。下一章将讨论数据库、关系数据库模型以及用于获取数据库内容的 SQL (结构化查询语言)。

16.7 因特网和万维网资源

pyxml.sourceforge.net

PyXML (一种 Python XML 处理包) 的主页。PyXML 包含一些有用的工具, 比如基于 DOM 和 SAX 的验证 XML 解析器。

4suite.org

4Suite (一种 Python XML 处理包) 的主页。4Suite 针对基于 DOM 的解析提供了几个 DOM 实现, 并为其他 XML 相关技术提供工具。

www.python.org/doc/current/lib/content-handler-objects.html

这个网站提供了 `xml.sax.ContentHandler` 事件处理程序的文档。

第 17 章 数据库应用程序编程接口 (DB-API)

学习目标

- 理解关系数据库模型
- 用 SQL 进行基本的数据库查询
- 利用 MySQLdb 模块的方法在数据库中查询、插入和更新数据

17.1 概述

第 14 章讨论了顺序访问和随机访问文件处理。前者适合对大部分或全部文件数据进行处理的应用程序，后者适合只对小部分文件数据进行处理的应用程序。例如在事务处理中，至关重要的一项能力便是快速定位和更新特定数据项。Python 支持这两种类型的文件处理。

“数据库”(Database)是集成的数据集合。许多公司都用数据库来组织员工信息，比如姓名、地址和电话号码等等。可采取多种策略来组织数据，以简化数据访问及处理。“数据库管理系统”(DBMS)采取与数据库的格式一致的方式来存储和组织数据。通过 DBMS 访问和存储数据时，不必关心数据库的内部表示。目前最流行的数据库系统是“关系数据库”，它们用表来存储数据，并定义了不同表之间的关系。“结构化查询语言”(SQL——要么按单独的字母发音，要么念作“sequel”)可用于几乎所有关系数据库系统，以便“查询”(请求与指定条件相符的信息)和处理数据。

流行的关系数据库系统包括 Microsoft SQL Server, Oracle, Sybase, DB2, Informix 和 MySQL 等等。本章的例子使用的是 MySQL 3.23.41。在 Deitel & Associates 网站 (www.deitel.com)，逐步讲解了如何安装 MySQL，并提供了一些有用的 MySQL 命令，以便创建、填充和删除表。

程序语言通过一个“接口”(简化数据库管理系统和程序通信的一种软件)来连接关系数据库并与之交互。Python 程序员使用遵循 Python DB-API (数据库应用程序编程接口) 规范的模块与数据库通信。17.5 节讨论了 DB-API 规范。

17.2 关系数据库模型

关系数据库模型是数据的逻辑表现形式，它允许在不牵涉数据物理结构的前提下考虑数据间的关系。关系数据库由一系列表组成。图 17.1 展示了一个可用于人事系统的表，名为 Employee，主要用于维护各个员工的特定属性。一个特定的表行称为“记录”(或“行”)。该表含有 6 条记录。每条记录的 number “字段”(或“列”)是对表中的数据进行引用的“主键”。主键是表中的一个字段或者数个字段的组合，其中包含不重复的、独一无二的数字。这样就可根据至少一个与众不同的值来标识每条记录。主键字段的例子包括社会安全号、员工 ID 以及库存系统中的零件号码。图 17.1 的记录按主键“排序”。在这个例子中，记录按升序排列(也可按降序)。

表中每列都代表一个不同的字段。记录本身在表中通常是不重复的(由主键标识)，但特定字段的值可能在多条记录中重复。例如，Employee 表有 3 条记录的 Department 字段都包含数字 413。

通常，数据库的不同用户对于数据和数据间的关系有着不同要求。有的用户只需要表列的一个子集。为获取子集，应用 SQL 语句从表中“选择”特定的数据。SQL 提供完备的命令集(包括 SELECT)，允许程序员定义复杂的“查询”，以便从表中选择数据。查询结果通常称为“结果集”(或“记录集”)。例如，可从图 17.1 的表中选择数据，从而新建一个结果集，其中只包含每个部门所在的地点。结果集如图 17.2 所示。17.4 节将详细讨论 SQL 查询。

Number	Name	Department	Salary	Location
23603	Jones	413	1100	New Jersey
24568	Kerwin	413	2000	New Jersey
34589	Larson	642	1800	Los Angeles
35761	Myers	611	1400	Orlando
47132	Neumann	413	9000	New Jersey
78321	Stephens	611	8500	Orlando

记录 / 行 {

主键 {

字段 / 列 {

图 17.1 Employee 表的关系数据库结构

Department	Location
413	New Jersey
611	Orlando
642	Los Angeles

图 17.2 从 Employee 表选择 Department 和 Location 数据而生成的结果集

17.3 关系数据库简介: Books 数据库

本节用一个示范数据库 Books 来概述 SQL。讨论 SQL 之前,要先解释一下 Books 数据库包含的表。注意,可在 Deitel 网站的下载网页 (www.deitel.com/books/downloads.html) 下载本书所有示例的源代码。其中包括一个 DBSetup.py 程序,可用它创建 Books 数据库,并在其中填充示范数据。

我们将通过 Books 数据库介绍各种数据库概念,比如如何用 SQL 从数据库获得有用的信息,以及如何操纵数据库。解开从网上下载的示例程序,在与本章对应的目录中,可以找到这个数据库。注意在 Windows 上使用 MySQL 时,数据库名称不用区分大小写(也就是说,Books 和 books 指的是同一个数据库)。但在 Linux 上,数据库名称必须区分大小写(Books 和 books 是两个不同的数据库)。

该数据库由 4 个表构成: Authors, Publishers, AuthorISBN 和 Titles。Authors 表(参见图 17.3)包括 3 个字段(列),分别维护每个作者的 ID 号、名字和姓氏。图 17.4 展示了该表的示范数据。

字段	说明
AuthorID	作者 ID 号。在 Books 数据库中,这个 int 字段定义成“自增字段”。在表中插入每一条新记录时,数据库都会使 AuthorID 值自增,保证每条记录都有独一无二的 AuthorID。这是该表的主键
FirstName	作者的名字(一个字符串)
LastName	作者的姓氏(一个字符串)

图 17.3 Books 数据库的 Authors 表

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tom	Nieto
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin

AuthorID	FirstName	LastName
7	Praveen	Sadhu
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield

图 17.4 Books 数据库的 Authors 表的数据

Publishers 表（如图 17.5 所示）由两个字段构成，代表每个出版商的 ID 及其名称。图 17.6 展示了该表的示范数据。

字段	说明
PublisherID	出版商的 ID 号。这个自增的 int 字段是表的主键
PublisherName	出版商的名称（一个字符串）

图 17.5 Books 数据库的 Publishers 表

PublisherID	PublisherName
1	Prentice Hall
2	Prentice Hall PTG

图 17.6 Books 数据库的 Publishers 表的数据

AuthorISBN 表（参见图 17.7）由两个字段构成，分别维护作者的 ID 号及其书籍的 ISBN 号。这个表有助于将作者姓名与其作品的标题对应起来。图 17.8 是这个表的一部分示范数据。ISBN 是指“国际标准图书编号”，这是一种全球统一的编号方案，保证每本书都具有不重复的标识号（注意，为节省空间，图 17.8 分两栏显示，每一栏都包含了 AuthorID 和 ISBN 字段）。

字段	说明
AuthorID	作者的 ID 号，便于数据库将书和特定的作者对应起来。这个字段的整数 ID 号必须同时出现在 Authors 表中
ISBN	一本书的 ISBN 号（一个字符串）

图 17.7 Books 数据库的 AuthorISBN 表

AuthorID	ISBN	AuthorID	ISBN
1	0130895725	1	0130284181
1	0132261197	1	0130895601
1	0130895717	2	0130895725
1	0135289106	2	0132261197
1	0139163050	2	0130895717
1	013028419x	2	0135289106
1	0130161438	2	0139163050
1	0130856118	2	013028419x
1	0130125075	2	0130161438
1	0138993947	2	0130856118
1	0130852473	2	0130125075

AuthorID	ISBN	AuthorID	ISBN
1	0130829277	2	0138993947
1	0134569555	2	0130852473
1	0130829293	2	0130829277
1	0130284173	2	0134569555
2	0130829293	3	0130856118
2	0130284173	3	0134569555
2	0130284181	3	0130829293
2	0130895601	3	0130284173
3	013028419x	3	0130284181
3	0130161438	4	0130895601

图 17.8 Books 的 AuthorISBN 表的数据 (只显示一部分)

Titles 表 (如图 17.9 所示) 由 7 个字段构成, 用于在数据库中维护书籍的常规信息。这些信息包括每本书的 ISBN 号、标题、版次、版权年和出版商 ID 号; 另外还有一个文件名, 该文件包含了书籍的封面图像; 最后是每本书的定价。图 17.10 展示了 Titles 表的示范数据。

字段	说明
ISBN	书的 ISBN 号 (字符串)
Title	书的标题 (字符串)
EditionNumber	书的版次 (字符串)
Copyright	书的版权年 (int 值)
PublisherID	出版商的 ID 号 (int 值)。这个值对应于 Publishers 中的 ID 号
ImageFile	封面图像文件名 (字符串)
Price	书的建议零售价 (实数)。注意, 本数据库显示的定价只用作示范

图 17.9 Books 数据库的 Titles 表

ISBN	Title	EditionNumber	PublisherID	Copyright	ImageFile	Price
0130923613	Python How to Program	1	1	2002	python.jpg	\$69.95
0130622214	C# How to Program	1	1	2002	cshtp.jpg	\$69.95
0130341517	Java How to Program	4	1	2002	jhtp4.jpg	\$69.95
0130649341	The Complete Java Training Course	4	2	2002	javactc4.jpg	\$109.95
0130895601	Advanced Java 2 Platform How to Program	1	1	2002	advjhtp1.jpg	\$69.95
0130308978	Internet and World Wide Web How to Program	2	1	2002	iw3htp2.jpg	\$69.95
0130293636	Visual Basic .NET How to Program	2	1	2002	vbnet.jpg	\$69.95
0130895636	The Complete C++ Training Course	3	2	2001	cppctc3.jpg	\$109.95
0130895512	The Complete e-Business & e-Commerce Programming Training Course	1	2	2001	ebecctc.jpg	\$109.95
013089561X	The Complete Internet & World Wide Web Programming Training Course	2	2	2001	iw3ctc2.jpg	\$109.95
0130895547	The Complete Perl Training Course	1	2	2001	perl.jpg	\$109.95
0130895563	The Complete XML Programming Training Course	1	2	2001	xmlctc.jpg	\$109.95
0130895725	C How to Program	3	1	2001	chtp3.jpg	\$69.95
0130895717	C++ How to Program	3	1	2001	cpphtp3.jpg	\$69.95

ISBN	Title	EditionNumber	PublisherID	Copyright	ImageFile	Price
013028419X	e-Business and e-Commerce How to Program	1	1	2001	ebechtp1.jpg	\$69.95
0130622265	Wireless Internet and Mobile Business How to Program	1	1	2001	wireless.jpg	\$69.95
0130284181	Perl How to Program	1	1	2001	perlhtp1.jpg	\$69.95
0130284173	XML How to Program	1	1	2001	xmlhtp1.jpg	\$69.95
0130856118	The Complete Internet and World Wide Web Programming Training Course	1	2	2000	iw3ctcl.jpg	\$109.95
0130125075	Java How to Program (Java 2)	3	1	2000	jhtp3.jpg	\$69.95
0130852481	The Complete Java 2 Training Course	3	2	2000	javactc3.jpg	\$109.95
0130323640	e-Business and e-Commerce for Managers	1	1	2000	ebecm.jpg	\$69.95
0130161438	Internet and World Wide Web How to Program	1	1	2000	iw3htp1.jpg	\$69.95
0130132497	Getting Started with Visual C++ 6 with an Introduction to MFC	1	1	1999	gsvc.jpg	\$49.95
0130829293	The Complete Visual Basic 6 Training Course	1	2	1999	vbctcl.jpg	\$109.95
0134569555	Visual Basic 6 How to Program	1	1	1999	vbhtp1.jpg	\$69.95
0132719746	Java Multimedia Cyber Classroom	1	2	1998	javactc.jpg	\$109.95
0136325890	Java How to Program	1	1	1998	jhtp1.jpg	\$69.95
0139163050	The Complete C++ Training Course	2	2	1998	cppctc2.jpg	\$109.95
0135289106	C++ How to Program	2	1	1998	cpphtp2.jpg	\$49.95
0137905696	The Complete Java Training Course	2	2	1998	javactc2.jpg	\$109.95
0130829277	The Complete Java Training Course (Java 1.1)	2	2	1998	javactc2.jpg	\$99.95
0138993947	Java How to Program (Java 1.1)	2	1	1998	jhtp2.jpg	\$49.95
0131173340	C++ How to Program	1	1	1994	cpphtp1.jpg	\$69.95
0132261197	C How to Program	2	1	1994	chtp2.jpg	\$49.95
0131180436	C How to Program	1	1	1992	chtp.jpg	\$69.95

图 17.10 Books 数据库的 Titles 表的数据

图 17.11 展示了 Books 数据库各个表的关系。第一行是表名。名字以斜体显示的字段包含该表的主键。主键唯一性地标识了表中的每条记录。每条记录的主键字段都必须赋值，而且这个值不能重复。这称为“实体完整性规则”。注意，AuthorISBN 表包含两个以斜体显示的字段。这表明要由两个字段构成“复合主键”（表中的每条记录都必须有一个独一无二的 AuthorID-ISBN 组合。例如，几条记录的 AuthorID 都可能为 2，而且几条记录的 ISBN 都可能为 0130895601，但是只能有一条记录的 AuthorID 为 2，ISBN 为 0130895601。

常见编程错误 17.1 忘记为每条记录的主键字段提供一个值会违反实体完整性规则，DBMS 会报错。

常见编程错误 17.2 为多条记录的主键字段提供重复的值，DBMS 会报错。

在图 17.11 中，不同表之间的联接线指明表和表的关系。先来看看 Publishers 和 Titles 表之间的联接线。在这条线上，Publishers 表那一端有一个数字 1，Titles 端则有一个无穷大 (∞) 符号。这条联接线表示“一对多”关系，即 Publishers 表中的每个出版商都可与 Titles 表中任意数量的书对应。注意，这条联接线将 Publishers 表的 PublisherID 字段与 Titles 表的 PublisherID 字段链接起来。在 Titles 表中，PublisherID 字段是“外键”，它引用了另一个表的主键（本例是引用 Publishers 表中的 PublisherID 字段）。程序员要在创建表时指定外键。外键有助于维持“引用完整性规则”：每个外键字段的值都必须出现在另一个表的主键中。外键允许来自多个表的信息“联接”到一起，以便进行分析。主键及其对应的外键之

间存在“一对多”关系。这意味着外键字段值可在它自己的表中多次出现，但在另一个表中作为主键使用时，则只能出现一次。表之间的联接线表示一个表的外键和另一个表的主键之间的联接。

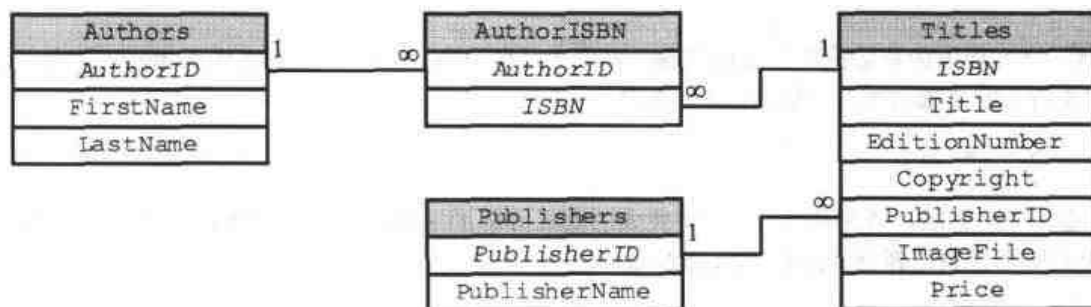


图 17.11 Books 数据库中各个表的关系

常见编程错误 17.3 如果外键值在另一个表中不是主键值，会违反“引用完整性规则”，DBMS 会报错。

AuthorISBN 和 Authors 表之间的联接线表明，针对 Authors 表中的每个作者，AuthorISBN 都可包含由其编著的、任意数量的书的 ISBN 号。AuthorISBN 表的 AuthorID 字段是 Authors 表的 AuthorID (主键) 字段的一个外键。同样请注意，这条线将 AuthorISBN 表中的外键与 Authors 表中相应的主键链接起来。AuthorISBN 表链接了 Titles 和 Authors 表的信息。

Titles 和 AuthorISBN 表之间的联接线展示了另一个“一对多”关系：一本书可以有多个作者。事实上，AuthorISBN 表唯一的用途就是表示 Authors 和 Titles 表之间的多对多关系；作者可以写任意数量的书，一本书可由任意数量的作者编著。

17.4 结构化查询语言 (SQL)

本节以 Books 示范数据库为例概述了结构化查询语言 (SQL)。这些 SQL 查询为本章例子所用的 SQL 奠定了基础。

图 17.12 列出了 SQL 关键字，并分别说明了它们的含义。在后续几小节，将以完整的 SQL 查询为例来讨论这些 SQL 关键字。虽然还有其他 SQL 关键字，但它们超出了本书范围。

SQL 关键字	说明
SELECT	从一个或多个表选择 (获取) 字段
FROM	指定从中获取字段或删除记录的表。所有 SELECT 和 DELETE 语句都需要该关键字
WHERE	指定按什么条件获取行
INNER JOIN	联接多个表的记录，提供单个记录集
GROUP BY	指定记录分组条件
ORDER BY	指定记录排序条件
INSERT	将数据插入指定的表
UPDATE	更新指定表中的数据
DELETE	从指定的表删除数据

图 17.12 SQL 查询关键字

17.4.1 基本的 SELECT 查询

下面来看从 Books 数据库提取信息的几个 SQL 查询。一个典型的 SQL 查询可从数据库的一个或多

个表中“选择”信息。这样的选择由“SELECT 查询”执行。它的基本格式是：

```
SELECT * FROM tableName
```

其中，星号（*）指出应从数据库的 *tableName* 表中选择所有列。例如，要想选择 Authors 表的全部内容（即图 17.4 所示的全部数据），请使用以下查询：

```
SELECT * FROM Authors
```

要从表中选择特定字段，请将星号（*）替换成用逗号分隔的字段名列表。例如，使用以下查询可只从 Authors 表的所有行中选择 AuthorID 和 LastName 字段：

```
SELECT AuthorID, LastName FROM Authors
```

这个查询只返回图 17.13 所示的数据。在这个结果集中，各个列遵循查询所指定的顺序。注意，假如一个字段名包含空格，整个字段名必须在查询中用方括号（[]）封闭。假定一个字段名为 First Name，在查询中应写为 [First Name]。

AuthorID	LastName	AuthorID	LastName
1	Deitel	8	McPhie
2	Deitel	9	Yaeger
3	Nieto	10	Zlatkina
4	Steinbuhler	12	Wiedermann
5	Santry	12	Liperi
6	Lin	13	Listfield
7	Sadhu		

图 17.13 Authors 表的 AuthorID 和 LastName

常见编程错误 17.4 SQL 语句用星号（*）选择字段时，不一定每次都按相同顺序返回那些字段。如字段顺序在表中发生改变，结果集中的字段顺序也会相应的改变。

性能提示 17.1 如果程序不知道结果集中的字段顺序，就必须按名字来处理字段。这可能要求在结果集中对字段名进行线性搜索。如果用户指定了要从表中选择的字段名，接收结果集的应用程序便事先知道了字段顺序。这样可更高效地处理数据，因为可按列号直接访问字段。

17.4.2 WHERE 子句

大多数时候，用户要在数据库中查找符合特定“选择条件”的记录。只有与条件相匹配的记录才会被选中。在 SELECT 查询中，SQL 使用可选的 WHERE 子句来指定选择条件。以下是含有选择条件的最简单的 SELECT 查询形式：

```
SELECT fieldName1, fieldName2, ... FROM tableName WHERE criteria
```

例如，要从 Titles 表选择 Copyright 年在 1999 之后的书的 Title, Edition 和 Copyright 字段，就应使用以下查询：

```
SELECT Title, EditionNumber, Copyright
FROM Titles
WHERE Copyright > 1999
```

图 17.14 显示了上述查询的结果集。注意，在构建一个要在 Python 中使用的查询时，需要创建一个包含

完整查询的字符串。但在书中显示查询时，则采用多行和缩进格式，以增强可读性。

性能提示 17.2 使用选择条件可改进性能，因为这种查询往往只选择数据库的一小部分。显然，处理小部分数据的效率要高于处理数据库全部数据的效率。

Title	EditionNumber	Copyright
Internet and World Wide Web How to Program	2	2002
Java How to Program	4	2002
The Complete Java Training Course	4	2002
The Complete e-Business & e-Commerce Programming Training Course	1	2001
The Complete Internet & World Wide Web Programming Training Course	2	2001
The Complete Perl Training Course	1	2001
The Complete XML Programming Training Course	1	2001
C How to Program	3	2001
C++ How to Program	3	2001
The Complete C++ Training Course	3	2001
e-Business and e-Commerce How to Program	1	2001
Internet and World Wide Web How to Program	1	2000
The Complete Internet and World Wide Web Programming Training Course	1	2000
Java How to Program (Java 2)	3	2000
The Complete Java 2 Training Course	3	2000
XML How to Program	1	2001
Perl How to Program	1	2001
Advanced Java 2 Platform How to Program	1	2002
e-Business and e-Commerce for Managers	1	2000
Wireless Internet and Mobile Business How to Program	1	2001
C# How To Program	1	2002
Python How to Program	1	2002
Visual Basic .NET How to Program	2	2002

图 17.14 从 Titles 表选择版权年在 1999 年之后的书名

WHERE 子句的条件可包含运算符 <、>、<=、>=、=、<> 和 LIKE。LIKE 用于“模式匹配”，可在其中使用百分号 (%) 和下划线 (_) 这两个通配符。模式匹配使 SQL 能搜索“与模式匹配”的字符串。

如模式中包含百分号 (%), 就表明在百分号位置，有零个或多个指定的字符。例如，以下查询可搜索姓氏以字母 D 开头的作者记录：

```
SELECT AuthorID, FirstName, LastName
FROM Authors
WHERE LastName LIKE 'D%'
```

以上查询会选择图 17.15 所示的两条记录，因为数据库中有两个作者的姓氏以字母 D 开头（后续零个或多个字符）。在 WHERE 子句的 LIKE 模式中，% 表明在 LastName 字段的字母 D 之后，可以有任意数量的字符。注意，模式字符串用一对单引号封闭。

移植性提示 17.1 并非所有数据库系统都支持 LIKE 运算符，所以要想使用它，必须阅读数据库系统的文档，弄清楚这一点。

移植性提示 17.2 有的数据库在 LIKE 表达式中使用的不是 %，而是 *。

移植性提示 17.3 在某些数据库中，字符串数据要区分大小写。

移植性提示 17.4 在某些数据库中，表名和字段名要区分大小写。

良好编程习惯 17.1 习惯上，在不区分大小写的系统上，SQL 关键字要全部采用大写字母。这样可在 SQL 语句中突出显示 SQL 关键字。

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel

图 17.15 从 Authors 表选择姓氏以 D 开头的作者

假如模式字符串中包含下划线（_）字符，表明在符合条件的字符串中，只能有一个字符占据下划线的位置。例如，在以下查询选择的记录中，作者姓氏要以任何字符开头（用_指定），后跟字母 i 和任意数量的其他字符（用%指定）：

```
SELECT AuthorID, FirstName, LastName
FROM Authors
WHERE LastName LIKE '_i%'
```

以上查询会选择如图 17.16 所示的记录。在数据库中，其姓氏中第二个字母是 i 的作者有 5 个。

AuthorID	FirstName	LastName
3	Tem	Nieto
6	Ted	Lin
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield

图 17.16 从 Authors 表选择其姓氏中第二个字母是 i 的作者

17.4.3 ORDER BY 子句

使用可选的 ORDER BY 子句，查询结果可按升序或降序排列。ORDER BY 子句的最简形式是：

```
SELECT fieldName1, fieldName2, ... FROM tableName ORDER BY field ASC
SELECT fieldName1, fieldName2, ... FROM tableName ORDER BY field DESC
```

ASC 代表升序（最低到最高），DESC 代表降序（最高到最低），而 *field* 的值决定了要排序的字段。

例如，为了获得一个作者列表，并使姓氏升序排列（图 17.17），请使用以下查询：

```
SELECT AuthorID, FirstName, LastName
FROM Authors
ORDER BY LastName ASC
```

注意默认是升序，所以 ASC 可选。要获得同一个列表，但姓氏降序排列（图 17.18），请使用以下查询：

```
SELECT AuthorID, FirstName, LastName
FROM Authors
ORDER BY LastName DESC
```

AuthorID	FirstName	LastName
2	Paul	Deitel
1	Harvey	Deitel
6	Ted	Lin
12	Jonathan	Liperi
13	Jeffrey	Listfield
8	David	McPhie
3	Tem	Nieto
7	Praveen	Sadhu
5	Sean	Santry
4	Kate	Steinbuhler
11	Ben	Wiedermann
9	Cheryl	Yaeger
10	Marina	Zlatkina

图 17.17 让 Authors 表中的作者按 LastName 升序排列

AuthorID	FirstName	LastName
10	Marina	Zlatkina
9	Cheryl	Yaeger
11	Ben	Wiedermann
4	Kate	Steinbuhler
5	Sean	Santry
7	Praveen	Sadhu
3	Tem	Nieto
8	David	McPhie
13	Jeffrey	Listfield
12	Jonathan	Liperi
6	Ted	Lin
2	Paul	Deitel
1	Harvey	Deitel

图 17.18 让 Authors 表中的作者按 LastName 降序排列

ORDER BY 子句还可按多个字段对记录排序。这种查询的形式为：

```
ORDER BY field1 sortingOrder, field2 sortingOrder, ...
```

其中, *sortingOrder* 可设为 ASC 或 DESC。注意, 每个字段的 *sortingOrder* 不一定相同。例如以下查询:

```
SELECT AuthorID, FirstName, LastName
FROM Authors
ORDER BY LastName, FirstName
```

会先按姓氏升序排列所有作者, 再按名字排列。这样一来, 同姓作者的记录会按名字排序 (图 17.19)。

查询中可组合 WHERE 和 ORDER BY 子句。例如, 以下查询:

```
SELECT ISBN, Title, EditionNumber, Copyright, Price
FROM Titles
WHERE Title
LIKE '*How to Program' ORDER BY Title ASC
```

会从 Titles 表中返回 Title 以 "How to Program" 结尾的每本书的 ISBN、标题（书名）、版次、版权年以及定价；同时，结果集中的记录按 Title 升序排列。图 17.20 是查询结果。

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
6	Ted	Lin
12	Jonathan	Liperi
13	Jeffrey	Listfield
8	David	McPhie
3	Tem	Nieto
7	Praveen	Sadhu
5	Sean	Santry
4	Kate	Steinhilber
11	Ben	Wiedermann
9	Cheryl	Yaeger
10	Marina	Zlatkina

图 17.19 让 Authors 表中的作者先按 LastName 升序排列，再按 FirstName 排列

ISBN	Title	EditionNumber	Copyright	Price
0130895601	Advanced Java 2 Platform How to Program	1	2002	\$69.95
0131180436	C How to Program	1	1992	\$69.95
0130895725	C How to Program	3	2001	\$69.95
0132261197	C How to Program	2	1994	\$49.95
0130622214	C# How To Program	1	2002	\$69.95
0135289106	C++ How to Program	2	1998	\$49.95
0131173340	C++ How to Program	1	1994	\$69.95
0130895717	C++ How to Program	3	2001	\$69.95
013028419X	e-Business and e-Commerce How to Program	1	2001	\$69.95
0130308978	Internet and World Wide Web How to Program	2	2002	\$69.95
0130161438	Internet and World Wide Web How to Program	1	2000	\$69.95
0130341517	Java How to Program	4	2002	\$69.95
0136325890	Java How to Program	1	1998	\$49.95
0130284181	Perl How to Program	1	2001	\$69.95
0130923613	Python How to Program	1	2002	\$69.95
0130293636	Visual Basic .NET How to Program	2	2002	\$69.95
0134569555	Visual Basic 6 How to Program	1	1999	\$69.95
0130622265	Wireless Internet and Mobile Business How to Program	1	2001	\$69.95
0130284173	XML How to Program	1	2001	\$69.95

图 17.20 从 Titles 表返回书名以 "How to Program" 结尾的记录，并按 Title 升序排列

17.4.4 合并多个表的数据：INNER JOIN

数据库设计人员经常将相关数据分离到单独的表中，目的是避免在数据库中存储冗余数据。例如在

Books 数据库中包含 Authors 表和 Titles 表。我们用 AuthorISBN 表在作者及其著作之间提供“链接”。如果不将信息分离到单独的表中，就需要为 Titles 表中的每一项都包括作者信息。这会导致在数据库中存储重复的作者信息，因为同一个作者可能写了好几本书。

为进行分析，经常需要将多个表的数据合并成单个数据集，这称为表的“联接”。我们在 SELECT 查询中使用一个 INNER JOIN (内联) 操作来进行联接。内联操作会检测不同表的公共字段的值，对两个或更多表的记录进行合并。INNER JOIN 的最简形式是：

```
SELECT fieldName1, fieldName2, ...
FROM table1
INNER JOIN table2
ON table1.fieldName = table2.fieldName
```

INNER JOIN 子句的 ON 部分指定了要对每个表的哪些字段进行比较，从而确定最终联接的记录。例如，以下查询生成一个作者列表，并附带每个作者写的书的 ISBN 号：

```
SELECT FirstName, LastName, ISBN
FROM Authors
INNER JOIN AuthorISBN
ON Authors.AuthorID = AuthorISBN.AuthorID
ORDER BY LastName, FirstName
```

以上查询将 Authors 表的 FirstName 和 LastName 字段与 AuthorISBN 表的 ISBN 字段合并，同时按照 LastName 和 FirstName 字段进行升序排序。注意，在 INNER JOIN 的 ON 部分使用的 *tableName.fieldName* 语法。这种语法称为“完全限定名”(Fully Qualified Name)，它指定对表进行联接时，应该比较的每个表的字段。如果两个表的字段同名，必须采用“*tableName.*”语法。在任何查询中，都可采用同样的语法区分不同表中的同名字段。如果完全限定名以数据库名开头，还可用于执行跨越不同数据库的查询。

软件工程知识 17.1 如果 SQL 语句包含来自多个表的同名字段，就必须在字段名前附加表名前缀和一个点运算符（例如 Authors.AuthorID）。

常见编程错误 17.5 在查询中忘记为多个表中的同名字段使用完全限定名会导致出错。

查询中总可以包括一个 ORDER BY 子句。图 17.21 展示了上述查询的结果，它按 LastName 和 FirstName 进行排序。注意，为节省空间，这里的查询结果分两栏显示，每一栏都包含 FirstName、LastName 和 ISBN 字段。

FirstName	LastName	ISBN	FirstName	LastName	ISBN
Harvey	Deitel	0130895601	Harvey	Deitel	0130829293
Harvey	Deitel	0130284181	Harvey	Deitel	0134569555
Harvey	Deitel	0130284173	Harvey	Deitel	0130829277
Harvey	Deitel	0130852473	Paul	Deitel	0130125075
Harvey	Deitel	0138993947	Paul	Deitel	0130856118
Harvey	Deitel	0130856118	Paul	Deitel	0130161438
Harvey	Deitel	0130161438	Paul	Deitel	013028419x
Harvey	Deitel	013028419x	Paul	Deitel	0139163050
Harvey	Deitel	0139163050	Paul	Deitel	0130895601
Harvey	Deitel	0135289106	Paul	Deitel	0135289106
Harvey	Deitel	0130895717	Paul	Deitel	0130895717
Harvey	Deitel	0132261197	Paul	Deitel	0132261197
Harvey	Deitel	0130895725	Paul	Deitel	0130895725
Harvey	Deitel	0130125075	Tem	Nieto	0130284181
Paul	Deitel	0130284181	Tem	Nieto	0130284173

FirstName	LastName	ISBN	FirstName	LastName	ISBN
Paul	Deitel	0130284173	Tem	Nieto	0130829293
Paul	Deitel	0130829293	Tem	Nieto	0134569555
Paul	Deitel	0134569555	Tem	Nieto	0130856118
Paul	Deitel	0130829277	Tem	Nieto	0130161438
Paul	Deitel	0130852473	Tem	Nieto	013028419x
Paul	Deitel	0138993947			

图 17.21 合并 Authors 表中的作者及其书籍的 ISBN 号, 结果按 LastName 和 FirstName 升序排序

17.4.5 联接 Authors、AuthorISBN、Titles 和 Publishers 表的数据

Books 数据库包含一个预定义查询 (TitleAuthor), 它为数据库中的每本书选择标题、ISBN 号、作者名字、作者姓氏、版权年以及出版商名称。如果一本书有多个作者, 查询会为每个作者单独生成一条复合记录。TitleAuthor 查询如图 17.22 所示, 结果如图 17.23 所示。

```

1 SELECT Titles.Title, Titles.ISBN, Authors.FirstName,
2     Authors.LastName, Titles.Copyright,
3     Publishers.PublisherName
4 FROM
5     ( Publishers INNER JOIN Titles
6       ON Publishers.PublisherID = Titles.PublisherID )
7   INNER JOIN
8     ( Authors INNER JOIN AuthorISBN
9       ON Authors.AuthorID = AuthorISBN.AuthorID )
10  ON Titles.ISBN = AuthorISBN.ISBN
11 ORDER BY Titles.Title

```

图 17.22 Books 数据库的 TitleAuthor 查询

Title	ISBN	FirstName	LastName	Copyright	PublisherName
Advanced Java 2 Platform How to Program	0130895601	Paul	Deitel	2002	Prentice Hall
Advanced Java 2 Platform How to Program	0130895601	Harvey	Deitel	2002	Prentice Hall
Advanced Java 2 Platform How to Program	0130895601	Sean	Santry	2002	Prentice Hall
C How to Program	0131180436	Harvey	Deitel	1992	Prentice Hall
C How to Program	0131180436	Paul	Deitel	1992	Prentice Hall
C How to Program	0132261197	Harvey	Deitel	1994	Prentice Hall
C How to Program	0132261197	Paul	Deitel	1994	Prentice Hall
C How to Program	0130895725	Harvey	Deitel	2001	Prentice Hall
C How to Program	0130895725	Paul	Deitel	2001	Prentice Hall
C# How To Program	0130622214	Tem	Nieto	2002	Prentice Hall
C# How To Program	0130622214	Paul	Deitel	2002	Prentice Hall
C# How To Program	0130622214	Jeffrey	Listfield	2002	Prentice Hall
C# How To Program	0130622214	Cheryl	Yaeger	2002	Prentice Hall
C# How To Program	0130622214	Marina	Zlatkina	2002	Prentice Hall
C# How To Program	0130622214	Harvey	Deitel	2002	Prentice Hall
C++ How to Program	0130895717	Paul	Deitel	2001	Prentice Hall
C++ How to Program	0130895717	Harvey	Deitel	2001	Prentice Hall
C++ How to Program	0131173340	Paul	Deitel	1994	Prentice Hall
C++ How to Program	0131173340	Harvey	Deitel	1994	Prentice Hall

Title	ISBN	FirstName	LastName	Copyright	PublisherName
C++ How to Program	0135289106	Harvey	Deitel	1998	Prentice Hall
C++ How to Program	0135289106	Paul	Deitel	1998	Prentice Hall
e-Business and e-Commerce for Managers	0130323640	Harvey	Deitel	2000	Prentice Hall
e-Business and e-Commerce for Managers	0130323640	Kate	Steinbuhler	2000	Prentice Hall
e-Business and e-Commerce for Managers	0130323640	Paul	Deitel	2000	Prentice Hall
e-Business and e-Commerce How to Program	013028419X	Harvey	Deitel	2001	Prentice Hall
e-Business and e-Commerce How to Program	013028419X	Paul	Deitel	2001	Prentice Hall
e-Business and e-Commerce How to Program	013028419X	Tem	Nieto	2001	Prentice Hall

图 17.23 图 17.22 所示查询生成的部分结果集

图 17.22 的查询进行了缩进处理,以增强可读性。下面将这个查询分解成不同的部分。第 1~3 行包含用逗号分隔的字段列表,它们是查询需要返回的;从左到右的各个字段决定了结果集中的字段顺序。这个查询从 Titles 表中选择 Title 和 ISBN 字段,从 Authors 表选择 FirstName 和 LastName 字段,从 Titles 表选择 Copyright 字段,并从 Publishers 表选择 PublisherName 字段。为进行澄清,每个字段名都用它的表名进行完全限定(例如 Titles.ISBN)。

第 5~10 行指定对来自各个表的信息进行合并的 INNER JOIN 操作。有 3 个 INNER JOIN 操作。注意,虽然要求对两个表执行 INNER JOIN 操作,但每个表都可能是另一个查询或另一个 INNER JOIN 的结果。我们用圆括号来嵌套 INNER JOIN 操作;SQL 先对最内层的圆括号进行求值,然后对外层进行求值。首先是下面这个 INNER JOIN 操作:

```
( Publishers INNER JOIN Titles
  ON Publishers.PublisherID = Titles.PublisherID )
```

它联接 Publishers 表和 Titles 表,条件是每个表中的 PublisherID 字段相匹配。在结果生成的临时表中,包含了与每本书及其出版商有关的信息。

另一对嵌套的圆括号包含以下 INNER JOIN:

```
( Authors INNER JOIN AuthorISBN ON
  Authors.AuthorID = AuthorISBN.AuthorID )
```

它联接 Authors 表和 AuthorISBN 表,条件是每个表中的 AuthorID 字段相匹配。记住,对于多名作者共同写作的书籍的 ISBN 号,AuthorISBN 表含有多个条目。第 3 个 INNER JOIN 是:

```
( Publishers INNER JOIN Titles
  ON Publishers.PublisherID = Titles.PublisherID )
INNER JOIN
( Authors INNER JOIN AuthorISBN
  ON Authors.AuthorID = AuthorISBN.AuthorID )
  ON Titles.ISBN = AuthorISBN.ISBN
```

它对上述两个内联操作所生成的表进行联接,条件是第一个临时表的每条记录的 Titles.ISBN 字段与第二个临时表的每条记录的相应的 AuthorISBN.ISBN 字段匹配。所有这些 INNER JOIN 操作的结果就是一个临时表,可从中选择恰当的字段来生成整个查询的结果。最后,查询的第 11 行是:

```
ORDER BY Titles.Title
```

它指出所有记录都应按照 Title 进行默认的升序排列。

17.4.6 INSERT 语句

INSERT 语句在表中插入一条新记录。该语句最简单的形式是:

```
INSERT INTO tableName ( fieldName1, fieldName2, ... , fieldNameN )
VALUES ( value1, value2, ..., valueN )
```

其中, *tableName* 是要在其中插入记录的表。*tableName* 后面是一对圆括号, 其中包含一个用逗号分隔的字段名列表。字段名列表之后是 SQL 关键字 **VALUES**, 然后又是包括在圆括号内的一个用逗号分隔的值列表。不管顺序还是类型, 这个列表指定的值必须与表名之后列出的字段名匹配。例如, 假定将 *fieldName1* 指定为 *FirstName* 字段, 那么 *value1* 应该是代表名字的一个用单引号封闭的字符串。以下 **INSERT** 语句:

```
INSERT INTO Authors ( FirstName, LastName )
VALUES ( 'Sue', 'Smith' )
```

会在 *Authors* 表中插入一条记录。该语句首先指出要为 *FirstName* 和 *LastName* 字段插入值, 具体的值分别是 'Sue' 和 'Smith'。注意, 上述 SQL 语句没有指定 *AuthorID* 值, 因为 *AuthorID* 是 *Authors* 表中的一个“自增字段”。每次在该表添加一条新记录, MySQL 都会自动指派一个不重复的 *AuthorID* 值, 并按升序自动编号 (1, 2, 3, ……等等)。在这个例子中, MySQL 将 *AuthorID* 编号 14 指派给 Sue Smith。图 17.24 显示了 **INSERT INTO** 操作之后的 *Authors* 表。

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tom	Nieto
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin
7	Praveen	Sadhu
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield
14	Sue	Smith

图 17.24 插入记录之后的 *Authors* 表

常见编程错误 17.6 SQL 语句将单引号 (') 作为字符串的定界符。如果字符串本身包含一个单引号 (比如 O'Malley), 就必须用两个单引号来表示一个单引号 (例如 'O'Malley')。如果在 SQL 语句中没有像这样对单引号进行转义, 就会造成 SQL 语法错误。

17.4.7 UPDATE 语句

UPDATE 语句用于更新表中的数据。**UPDATE** 语句最简单的形式是:

```
UPDATE tableName
SET fieldName1 = value1, fieldName2 = value2, ... , fieldNameN = valueN
WHERE criteria
```

tableName 是要在其中更新记录的表名, 后续关键字 **SET**, 以及一个用逗号分隔的字段“名/值对”列表, 格式为 *fieldName = Value*。WHERE 子句指定以什么条件判断要更新的记录。例如以下 **UPDATE** 语句:

```
UPDATE Authors
SET LastName = 'Jones'
```

```
WHERE LastName = 'Smith' AND FirstName = 'Sue'
```

可更新 Authors 表中的一条记录。该语句指出, 针对目前 LastName 等于 Smith, 而且 FirstName 等于 Sue 的记录, 为 LastName 指派新值 Jones。如果在 UPDATE 操作之前已经知道 AuthorID (可能是因为以前搜索过记录), 那么 WHERE 子句可简化成:

```
WHERE AuthorID = 14
```

图 17.25 展示了执行以上 UPDATE 操作之后的 Authors 表。

常见编程错误 17.7 忘记为 UPDATE 语句使用一个 WHERE 子句会造成逻辑错误。

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tom	Nieto
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin
7	Praveen	Sadhu
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield
14	Sue	Jones

图 17.25 用 UPDATE 操作更改记录之后的 Authors 表

17.4.8 DELETE 语句

SQL 的 DELETE 语句用于从表中删除数据。DELETE 语句的最简形式是:

```
DELETE FROM tableName WHERE criteria
```

tableName 是指要从中删除记录的表名。WHERE 子句指定了用于判断具体要删除哪些记录的条件。例如以下 DELETE 语句:

```
DELETE FROM Authors
WHERE LastName = 'Jones' AND FirstName = 'Sue'
```

将从 Authors 表删除 Sue Jones 的记录。图 17.26 展示了这个 DELETE 操作之后的 Authors 表。

常见编程错误 17.8 WHERE 子句可匹配多条记录。从数据库中删除记录时, 必须准确定义 WHERE 子句, 让它只匹配确实要删除的记录。

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tom	Nieto

AuthorID	FirstName	LastName
4	Kate	Steinbuhler
5	Sean	Santry
6	Ted	Lin
7	Praveen	Sadhu
8	David	McPhie
9	Cheryl	Yaeger
10	Marina	Zlatkina
11	Ben	Wiedermann
12	Jonathan	Liperi
13	Jeffrey	Listfield

图 17.26 用 DELETE 操作删除记录之后的 Authors 表

17.5 Python DB-API 规范

本章的示范代码使用 MySQL 数据库系统，然而，Python 除了 MySQL 之外，还支持其他许多数据库。许多现成的模块可与大多数流行的数据库通信，从而向程序员隐藏数据库的细节。这些模块遵循的是 Python 的“应用程序编程接口”（DB-API）规范，它为针对任何数据库的操作都指定了公共的对象和方法名称。

具体说来，DB-API 描述了一个用于访问（连接）数据库的 Connection 对象。然后，程序可用 Connection 对象创建 Cursor 对象，该对象负责处理和接收数据。本章后文将通过实际的例子讨论这些对象的方法和属性。

利用 Cursor，程序既可对数据库执行操作，比如执行查询、在表中插入行、从表中删除行等等；也可处理查询返回的数据。可用 3 个方法从查询结果集中获取行，即 `fetchone`、`fetchmany` 和 `fetchall`。其中，`fetchone` 方法返回一个元组，其中包含存储在 Cursor 中一个结果集的下一行；`fetchmany` 方法需要指定一个参数（准备获取的行数），并以“元组的元组”形式，从结果集中返回后续一系列行；`fetchall` 方法则采用“元组的元组”的形式，返回结果集中的所有行。对于大型数据库，使用 `fetchall` 是不切实际的。

DB-API 的优点在于，程序无需对连接的数据库有更深入的认识。所以，只需在 Python 源代码中进行少量修改，即可让程序使用不同的数据库。例如，要从 MySQL 数据库切换成其他数据库，只需修改 3、4 行代码即可。但是，在不同数据库间切换时，可能需要修改 SQL 代码（如解决大小写敏感问题）。

17.6 数据库查询示例

图 17.27 展示了一个 CGI 程序，它对 Books 数据库执行简单查询。它获取与 Authors 表中的作者有关的所有信息，并用一个 XHTML 表格显示数据。该程序演示了如何连接数据库、查询数据库以及显示结果。在后面的讨论中，介绍了程序的一些关键的 DB-API 概念。注意，本例的 CGI 脚本适用于在 Microsoft Windows 上运行的 Apache Web 服务器。还可在 Deitel 网站的下载网页，下载本例的另一个版本，它适合在 Linux 上运行的 Apache 中使用。

第 6 行导入 MySQLdb 模块，其中包含可在 Python 中操纵 MySQL 数据库的类和函数（下载地址是 sourceforge.net/projects/mysql-python）。要想获得具体的安装指南，请访问 www.deitel.com。

第 86~105 行创建一个 XHTML 表单，用户可利用它指定如何对 Authors 表的记录进行排序。第 24~36 行取得并处理这个表单。记录将根据指派给变量 `sortBy` 的字段进行排序。默认情况下，记录将按 AuthorID 进行排序。选择一个单选按钮，就可依据另一个字段进行排序。类似地，变量 `sortOrder` 指定了是

升序还是降序, 它的值要么由用户指定, 要么默认为"ASC".

第 42 行创建名为 `connection` 的一个 `Connection` 对象, 以便管理程序和数据库之间的连接。`MySQLdb.connect` 函数通过关键字参数 `db` 获得数据库的名称, 并相应地创建连接。注意, 对于除 Windows 之外的其他操作系统, MySQL 可能需要一个用户名和密码才能连接数据库。如果是这样, 请将相应的值以字符串形式, 传给函数 `MySQLdb.connect` 的 `user` 和 `passwd` 这两个关键字参数。如果 `MySQLdb.connect` 函数执行失败, 会引发一个 `MySQLdb` 的 `OperationalError` 异常。

第 51 行调用 `Connection` 的 `cursor` 方法, 为数据库创建一个 `Cursor` 对象。`Cursor` 的 `execute` 方法以参数形式获得一个 SQL 命令, 并针对数据库执行该命令。第 54~55 行查询 `Authors` 表, 获取它的所有记录, 并按照 `sortBy` 指定的字段进行排序, 升序还是降序则由 `sortOrder` 的值决定。

`Cursor` 对象在内部存储着一个数据库查询的结果。`Cursor` 的 `description` 属性包含一个“元组的元组”, 每个元组都提供了与 `execute` 方法所获得的结果集中的一个字段有关的信息。每个字段元组的第一个值是字段名。第 57 行将字段名记录的元组指派给变量 `allFields`。`Cursor` 的 `fetchall` 方法返回一个“元组的元组”, 其中包含内部存储的所有结果, 这些结果是通过执行方法而获得的。在返回的元组中, 每个子元组都代表数据库中的一条记录, 记录中的每个元素都代表那条记录中的一个字段的值。第 58 行将相匹配的记录的元组指派给变量 `allRecords`。

`Cursor` 的 `close` 方法(第 61 行)关闭 `Cursor` 对象; 第 62 行使用 `Connection` 的 `close` 方法关闭 `Connection` 对象。这些方法会显式地关闭 `Cursor` 和 `Connection` 对象。尽管在程序终止时, 由于对象被销毁, 所以会自动执行对象的 `close` 方法, 但程序员最好还是在对象不再需要之后, 立即显式地将其关闭。

良好编程习惯 17.2 程序不再需要 `Cursor` 和 `Connection` 对象后, 用相应的 `close` 方法显式关闭它们。

程序剩余部分用于在 XHTML 表格中显示数据库结果。第 65~83 行使用 `for` 循环显示 `Authors` 表的字段。对于每个字段, 程序都显示那个字段的元组中的第一项(第 69~70 行)。第 75~83 行使用嵌套 `for` 循环, 为 `Authors` 表的每条记录都显示一个表行。外层 `for` 循环(第 78 行)遍历当前记录的每个字段, 分别用一个新单元格显示它们。

```

1  #!c:\python\python.exe
2  # Fig. 17.27: fig17_27.py
3  # Displays contents of the Authors table,
4  # ordered by a specified field.
5
6  import MySQLdb
7  import cgi
8  import sys
9
10 def printHeader( title ):
11     print """Content-type: text/html
12
13     <?xml version = "1.0" encoding = "UTF-8"?>
14     <!DOCTYPE html PUBLIC
15         "-//W3C//DTD XHTML 1.0 Transitional//EN"
16         "DTD/xhtml1-transitional.dtd">
17     <html xmlns = "http://www.w3.org/1999/xhtml"
18         xml:lang = "en" lang = "en">
19     <head><title>%s</title></head>
20
21     <body>""" % title
22
23     # obtain user query specifications
24     form = cgi.FieldStorage()
25
26     # get "sortBy" value
27     if form.has_key( "sortBy" ):
28         sortBy = form[ "sortBy" ].value
29     else:
30         sortBy = "firstName"
31
32     # get "sortOrder" value

```

```

33 if form.has_key( "sortOrder" ):
34     sortOrder = form[ "sortOrder" ].value
35 else:
36     sortOrder = "ASC"
37
38 printHeader( "Authors table from Books" )
39
40 # connect to database and retrieve a cursor
41 try:
42     connection = MySQLdb.connect( db = "Books" )
43
44 # error connecting to database
45 except MySQLdb.OperationalError, error:
46     print "Error:", error
47     sys.exit( 1 )
48
49 # retrieve cursor
50 else:
51     cursor = connection.cursor()
52
53 # query all records from Authors table
54 cursor.execute( "SELECT * FROM Authors ORDER BY %s %s" %
55     ( sortBy, sortOrder ) )
56
57 allFields = cursor.description # get field names
58 allRecords = cursor.fetchall() # get records
59
60 # close cursor and connection
61 cursor.close()
62 connection.close()
63
64 # output results in a table
65 print """\n<table border = "1" cellpadding = "3" >
66     <tr bgcolor = "silver" >""
67
68 # create table header
69 for field in allFields:
70     print "<td>%s</td>" % field[ 0 ]
71
72 print "</tr>"
73
74 # display each record as a row
75 for author in allRecords:
76     print "<tr>"
77
78     for item in author:
79         print "<td>%s</td>" % item
80
81     print "</tr>"
82
83 print "</table>"
84
85 # obtain sorting method from user
86 print ""
87     \n<form method = "post" action = "/cgi-bin/fig17_27.py">
88     Sort By:<br />""
89
90 # display sorting options
91 for field in allFields:
92     print """<input type = "radio" name = "sortBy"
93     value = "%s" />"" % field[ 0 ]
94     print field[ 0 ]
95     print "<br />"
96
97 print """<br />\nSort Order:<br />
98     <input type = "radio" name = "sortOrder"
99     value = "ASC" checked = "checked" />
100     Ascending
101     <input type = "radio" name = "sortOrder"
102     value = "DESC" />
103     Descending

```

```

104 <br /><br />\n<input type = "submit" value = "SORT" />
105 </form>\n\n</body>\n</html>"""

```

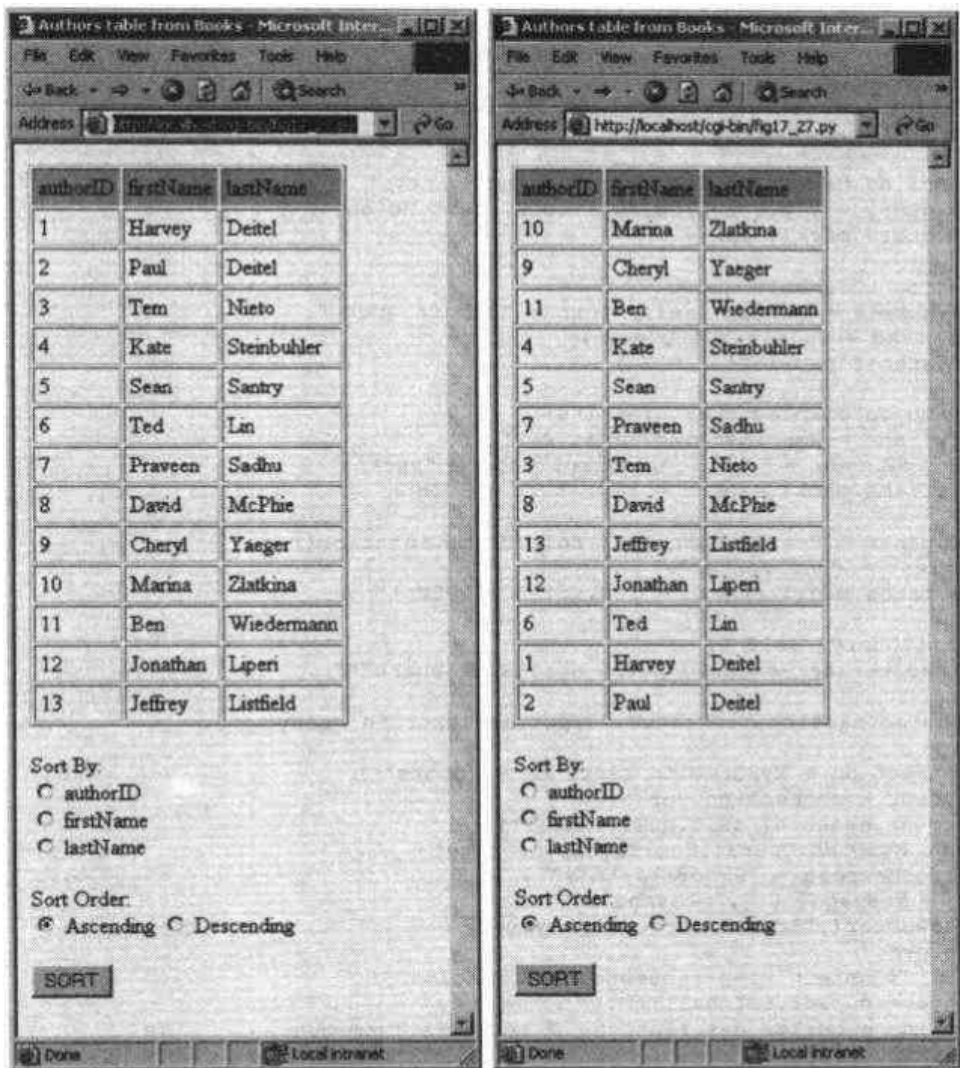


图 17.27 连接并查询数据库

17.7 查询 Books 数据库

图 17.28 对图 17.27 的例子进行了改进，它允许用户在 GUI 程序中输入任何查询。这个例子综合运用了数据库错误处理机制以及两个 Pmw 组件 (ScrolledFrame 和 PanedWidget)。Pmw 模块的详情已在第 11 章讲解。GUI 构造函数 (第 13~39 行) 创建了 4 个 GUI 元素。

```

1 # Fig. 17.28: fig17_28.py
2 # Displays results returned by a
3 # query on Books database.
4
5 import MySQLdb
6 from Tkinter import *
7 from tkMessageBox import *
8 import Pmw
9
10 class QueryWindow( Frame ):
11     """GUI Database Query Frame"""
12
13     def __init__( self ):

```

```

14     """QueryWindow Constructor"""
15
16     Frame.__init__( self )
17     Pmw.initialise()
18     self.pack( expand = YES, fill = BOTH )
19     self.master.title( \
20         "Enter Query, Click Submit to See Results." )
21     self.master.geometry( "525x525" )
22
23     # scrolled text pane for query string
24     self.query = Pmw.ScrolledText( self, text_height = 8 )
25     self.query.pack( fill = X )
26
27     # button to submit query
28     self.submit = Button( self, text = "Submit query",
29         command = self.submitQuery )
30     self.submit.pack( fill = X )
31
32     # frame to display query results
33     self.frame = Pmw.ScrolledFrame( self,
34         hscrollmode = "static", vscrollmode = "static" )
35     self.frame.pack( expand = YES, fill = BOTH )
36
37     self.panes = Pmw.PanedWidget( self.frame.interior(),
38         orient = "horizontal" )
39     self.panes.pack( expand = YES, fill = BOTH )
40
41     def submitQuery( self ):
42         """Execute user-entered query against database"""
43
44         # open connection, retrieve cursor and execute query
45         try:
46             connection = MySQLdb.connect( db = "Books" )
47             cursor = connection.cursor()
48             cursor.execute( self.query.get() )
49         except MySQLdb.OperationalError, message:
50             errorMessage = "Error %d:\n%s" % \
51                 ( message[ 0 ], message[ 1 ] )
52             showerror( "Error", errorMessage )
53             return
54         else: # obtain user-requested information
55             data = cursor.fetchall()
56             fields = cursor.description # metadata from query
57             cursor.close()
58             connection.close()
59
60             # clear results of last query
61             self.panes.destroy()
62             self.panes = Pmw.PanedWidget( self.frame.interior(),
63                 orient = "horizontal" )
64             self.panes.pack( expand = YES, fill = BOTH )
65
66             # create pane and label for each field
67             for item in fields:
68                 self.panes.add( iter( 0 ) )
69                 label = Label( self.panes.pane( item[ 0 ] ),
70                     text = item[ 0 ], relief = RAISED )
71                 label.pack( fill = X )
72
73             # enter results into panes, using labels
74             for entry in data:
75                 for i in range( len( entry ) ):
76                     label = Label( self.panes.pane( fields[ i ][ 0 ] ),
77                         text = str( entry[ i ] ), anchor = W,
78                         relief = GROOVE, bg = "white" )
79                     label.pack( fill = X )
80
81             self.panes.setnaturalsize()
82
83     def main():

```



```

85 QueryWindow().mainloop()
86
87 if __name__ == "__main__":
88     main()

```

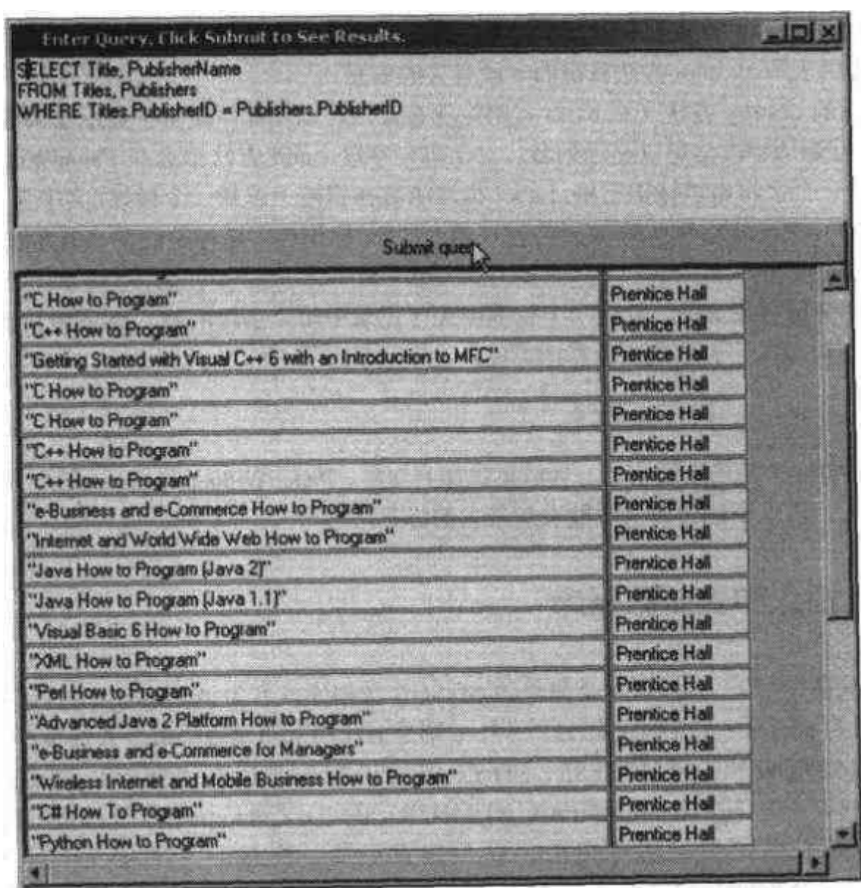


图 17.28 用于向数据库提交查询的 GUI 应用程序

程序显示由两个区域构成。顶部区域提供一个 ScrolledText 组件 (第 24~25 行), 可在其中输入查询字符串。text_height 属性将这个滚动文本区域的高度设为 8 行。一个 Button 组件 (第 28~30 行) 可调用方法, 针对数据库执行指定的查询字符串。

底部区域包括一个 ScrolledFrame 组件 (第 33~35 行), 用于显示查询结果。ScrolledFrame 组件是一种可滚动的显示区域。这里显示了水平和垂直滚动条, 我们将 hscrollmode 和 vscrollmode 属性的值设为 "static"。在 ScrolledFrame 中包含一个 PanedWidget 组件 (第 37~39 行), 它的作用是将结果记录分为不同的字段。Frame 的 interior 方法指出, PanedWidget 要在 ScrolledFrame 内部创建。PanedWidget 是进一步细分的帧, 即“窗格”。用户可改变这种窗格的大小。PanedWidget 构造函数的 orient 参数值可设为 "horizontal" 或 "vertical"。如果值是 "horizontal", 就表明窗格在帧内从左到右放置; 如果值是 "vertical", 就表明窗格在帧内从上到下放置。

一旦按下 Submit query (提交查询) 按钮, 就会由 submitQuery 方法 (第 41~82 行) 执行查询并显示结果。第 45~58 行包含一个 try/except/else 语句, 它连接并查询数据库。try 语句创建一个 Connection 对象和一个 Cursor 对象, 并用 Cursor 的 execute 方法执行用户输入的查询。如果指定的数据库不存在, MySQLdb.connect 函数 (第 46 行) 会失败。如果查询字符串包含一个 SQL 语法错误, Cursor 的 execute 方法 (第 48 行) 会失败。一旦失败, 它们都会引发 OperationalError 异常。第 49~53 行处理这个异常, 并调用 tkMessageBox 的 showerror 函数, 显示恰当的错误消息。

如果用户输入的查询字符串成功执行, 程序将取得查询结果。第 54~58 行的 else 子句将查询获得的记录指派给变量 data, 并将元数据指派给变量 fields。“元数据”是对数据进行描述的数据。例如, 用

于结果集的元数据可能包括字段名及字段类型。以下元数据：

```
fields = cursor.description
```

包含了与用户输入的查询的结果集有关的描述信息（第 56 行）。Cursor 的 description 属性包含一个“元组的元组”，它提供了和 execute 方法获得的字段有关的信息。

PanedWidget 的 destroy 方法（第 61 行）删除现有窗格，以便在新窗格中显示查询数据（第 62~64 行）。第 67~71 行遍历字段信息以显示列名。对于每个字段，add 方法都会在 PanedWidget 上添加一个窗格。该方法取得一个字符串以标识窗格。Label 构造函数在窗格上添加一个标签，其中包含字段名（relief 属性设为 RAISED）。PanedWidget 的 pane 方法（第 69 行）标识这个新标签的父。该方法取得窗格名称，并返回对那个窗格的一个引用。

第 74~80 行遍历每条记录，创建一个标签以包含记录中每个字段的值。pane 方法为每个标签都指定了恰当的父帧。对于以下表达式所获得的标签：

```
self.panes.pane( fields[ i ][ 0 ] )
```

它的名称是记录中第 i 个值的字段名。一旦结果添加到窗格，PanedWidget 的 setnaturalsize 方法将每个窗格的值设得足够大，以保证能在窗格中能看到最大的标签。

17.8 读取、插入和更新数据库

下一个例子（图 17.29）要操纵一个简单的 MySQL 数据库，名为 AddressBook（通讯簿）。该数据库包含一个表，名为 addresses。表中含有 11 个列，分别是 ID（用于通讯簿中每个人的惟一性整数 ID 号）、FIRST_NAME、LAST_NAME、ADDRESS、CITY、STATE_PROVINCE、POSTAL_CODE、COUNTRY、EMAIL_ADDRESS、HOME_PHONE 和 FAX_NUMBER。除 ID 之外的所有字段都是字符串。程序允许在数据库中插入新记录，更新现有记录，以及搜索记录。注意，可在 Deitel 网站的下载网页（www.deitel.com/books/downloads.html）下载本书所有示例的源代码。其中包括一个 DBSetup.py 程序，可用它创建一个空白的 AddressBook 数据库。

AddressBook 类使用 Button 和 Entry 组件获取和显示地址信息。构造函数为一个通讯簿条目创建一个字段列表（第 32~34 行）。第 38 行初始化字典数据成员 entries，用它容纳对 Entry 组件的引用。第 44~60 行的 for 循环遍历这个列表，为每个字段都创建一个 Entry 组件（第 47~48 行）。循环还为数据成员 entries 添加对 Entry 组件的一个引用。第 58~60 行为每个条目都创建一个键名——具体名称由条目的字段名决定。

```
1 # Fig. 17.29: fig17_29.py
2 # Inserts into, updates and searches a database
3
4 import MySQLdb
5 from Tkinter import *
6 from tkMessageBox import *
7 import Pmw
8
9 class AddressBook( Frame ):
10     """GUI Database Address Book Frame"""
11
12     def __init__( self ):
13         """Address Book constructor"""
14
15         Frame.__init__( self )
16         Pmw.initialise()
17         self.pack( expand = YES, fill = BOTH )
18         self.master.title( "Address Book Database Application" )
19
20         # buttons to execute commands
```

```

21 self.buttons = Pmw.ButtonBox( self, padx = 0 )
22 self.buttons.grid( columnspan = 2 )
23 self.buttons.add( "Find", command = self.findAddress )
24 self.buttons.add( "Add", command = self.addAddress )
25 self.buttons.add( "Update", command = self.updateAddress )
26 self.buttons.add( "Clear", command = self.clearContents )
27 self.buttons.add( "Help", command = self.help, width = 14 )
28 self.buttons.alignbuttons()
29
30
31 # list of fields in an address record
32 fields = [ "ID", "First name", "Last name",
33           "Address", "City", "State Province", "Postal Code",
34           "Country", "Email Address", "Home phone", "Fax Number" ]
35
36 # dictionary with Entry components for values, keyed by
37 # corresponding addresses table field names
38 self.entries = {}
39
40 self.IDEntry = StringVar() # current address id text
41 self.IDEntry.set( "" )
42
43 # create entries for each field
44 for i in range( len( fields ) ):
45     label = Label( self, text = fields[ i ] + ":" )
46     label.grid( row = i + 1, column = 0 )
47     entry = Entry( self, name = fields[ i ].lower(),
48                  font = "Courier 12" )
49     entry.grid( row = i + 1, column = 1,
50               sticky = W+E+N+S, padx = 5 )
51
52     # user cannot type in ID field
53     if fields[ i ] == "ID":
54         entry.config( state = DISABLED,
55                      textvariable = self.IDEntry, bg = "gray" )
56
57     # add entry field to dictionary
58     key = fields[ i ].replace( " ", "_" )
59     key = key.upper()
60     self.entries[ key ] = entry
61
62 def addAddress( self ):
63     """Add address record to database"""
64
65     if self.entries[ "LAST_NAME" ].get() != "" and \
66        self.entries[ "FIRST_NAME" ].get() != "":
67
68         # create INSERT query command
69         query = """INSERT INTO addresses (
70             FIRST_NAME, LAST_NAME, ADDRESS, CITY,
71             STATE_PROVINCE, POSTAL_CODE, COUNTRY,
72             EMAIL_ADDRESS, HOME_PHONE, FAX_NUMBER
73             ) VALUES ("" + \
74             "%s", " * 10 % \
75             ( self.entries[ "FIRST_NAME" ].get(),
76             self.entries[ "LAST_NAME" ].get(),
77             self.entries[ "ADDRESS" ].get(),
78             self.entries[ "CITY" ].get(),
79             self.entries[ "STATE_PROVINCE" ].get(),
80             self.entries[ "POSTAL_CODE" ].get(),
81             self.entries[ "COUNTRY" ].get(),
82             self.entries[ "EMAIL_ADDRESS" ].get(),
83             self.entries[ "HOME_PHONE" ].get(),
84             self.entries[ "FAX_NUMBER" ].get() )
85         query = query[ :-2 ] + ")"
86
87         # open connection, retrieve cursor and execute query
88         try:
89             connection = MySQLdb.connect( db = "AddressBook" )
90             cursor = connection.cursor()
91             cursor.execute( query )

```

```

92     except MySQLdb.OperationalError, message:
93         errorMessage = "Error %d:\n%s" % \
94             ( message[ 0 ], message[ 1 ] )
95         showerror( "Error", errorMessage )
96     else:
97         cursor.close()
98         connection.close()
99         self.clearContents()
100
101     else: # user has not filled out first/last name fields
102         showwarning( "Missing fields", "Please enter name" )
103
104 def findAddress( self ):
105     """Query database for address record and display results"""
106
107     if self.entries[ "LAST_NAME" ].get() != "":
108
109         # create SELECT query
110         query = "SELECT * FROM addresses " + \
111             "WHERE LAST_NAME = '" + \
112             self.entries[ "LAST_NAME" ].get() + "'"
113
114         # open connection, retrieve cursor and execute query
115         try:
116             connection = MySQLdb.connect( db = "AddressBook" )
117             cursor = connection.cursor()
118             cursor.execute( query )
119         except MySQLdb.OperationalError, message:
120             errorMessage = "Error %d:\n%s" % \
121                 ( message[ 0 ], message[ 1 ] )
122             showerror( "Error", errorMessage )
123             self.clearContents()
124         else: # process results
125             results = cursor.fetchall()
126             fields = cursor.description
127
128             if not results: # no results for this person
129                 showinfo( "Not found", "Nonexistent record" )
130             else: # display information in GUI
131                 self.clearContents()
132
133                 # display results
134                 for i in range( len( fields ) ):
135
136                     if fields[ i ][ 0 ] == "ID":
137                         self.IDEntry.set( str( results[ 0 ][ i ] ) )
138                     else:
139                         self.entries[ fields[ i ][ 0 ] ].insert(
140                             INSERT, str( results[ 0 ][ i ] ) )
141
142                 cursor.close()
143                 connection.close()
144
145     else: # user did not enter last name
146         showwarning( "Missing fields", "Please enter last name" )
147
148 def updateAddress( self ):
149     """Update address record in database"""
150
151     if self.entries[ "ID" ].get():
152
153         # create UPDATE query command
154         entryItems= self.entries.items()
155         query = "UPDATE addresses SET"
156
157         for key, value in entryItems:
158
159             if key != "ID":
160                 query += " %s='%s'," % ( key, value.get() )
161
162         query = query[ :-1 ] + " WHERE ID=" + self.IDEntry.get()

```

```

163
164     # open connection, retrieve cursor and execute query
165     try:
166         connection = MySQLdb.connect( db = "AddressBook" )
167         cursor = connection.cursor()
168         cursor.execute( query )
169     except MySQLdb.OperationalError, message:
170         errorMessage = "Error %d:\n%s" % \
171             ( message[ 0 ], message[ 1 ] )
172         showerror( "Error", errorMessage )
173         self.clearContents()
174     else:
175         showinfo( "database updated", "Database Updated." )
176         cursor.close()
177         connection.close()
178
179     else: # user has not specified ID
180         showwarning( "No ID specified", ""
181             You may only update an existing record.
182             Use Find to locate the record,
183             then modify the information and press Update."" )
184
185     def clearContents( self ):
186         """Clear GUI panel"""
187
188         for entry in self.entries.values():
189             entry.delete( 0, END )
190
191         self.IDEntry.set( "" )
192
193     def help( self ):
194         "Display help message to user"
195
196         showinfo( "Help", ""Click Find to locate a record.
197             Click Add to insert a new record.
198             Click Update to update the information in a record.
199             Click Clear to empty the Entry fields.\n"" )
200
201     def main():
202         AddressBook().mainloop()
203
204     if __name__ == "__main__":
205         main()

```

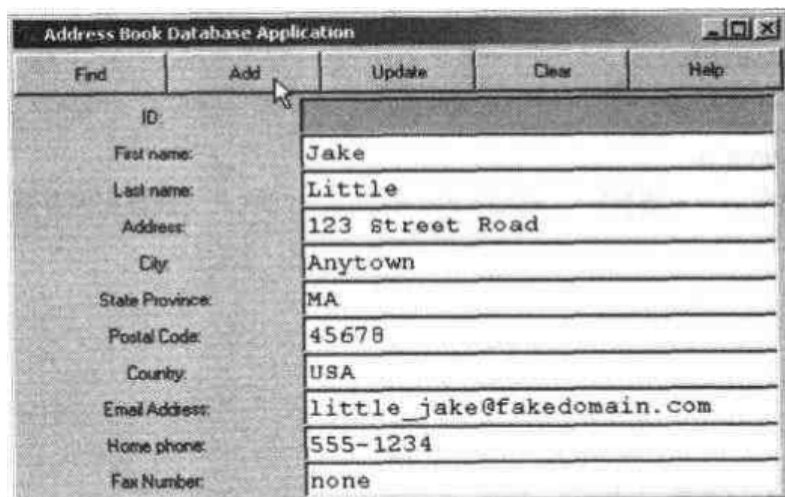


图 17.29 插入、查找和更新记录

`addRecord` 方法 (第 62~102 行) 在 `AddressBook` 数据库中添加一条新记录, 以响应 GUI 中的 `Add` 按钮。方法首先验证用户为名字和姓氏字段输入了值 (第 65~66 行)。如果用户为这些字段输入了值,

查询字符串会将一条记录插入数据库（第 69~85 行）。否则，`tkMessageBox` 的 `showwarning` 函数会提醒用户输入信息（第 101~102 行）。第 74 行包括 10 个字符串转义序列，它们的值由第 75~84 行包含的值进行替代。第 85 行结束 SQL 语句中的值列表的圆括号。

第 88~99 行包含一个 `try/except/else` 语句，它连接和更新数据库（也就是在数据库中插入新记录）。第 99 行调用 `clearContents` 方法（第 185~191 行），从而清除 GUI 的内容。如发生错误，就由 `tkMessageBox` 的 `showerror` 函数显示错误。

一旦用户在 GUI 中单击 Find 按钮，`findAddress` 方法（第 104~146 行）就会查询 `AddressBook` 数据库，以查找特定记录。第 107 行验证姓氏字段中是否包含数据。如果没有输入任何数据，程序会显示一个错误。如果用户在姓氏字段中输入了数据，一个 SQL 的 `SELECT` 语句会在数据库中搜索用户指定的姓氏。之所以在 `SELECT` 语句中使用星号（*），是因为第 126 行使用元数据来获取字段名。第 115~143 行包含一个 `try/except/else` 语句，它负责连接和查询数据库。如果这些操作成功执行，程序会从数据库收到结果（第 125~126 行）。如果查询没有获得任何结果，就会用一条消息提醒用户（第 128~129 行）。如查询得到了结果，第 134~140 行会在 GUI 中显示结果。每个字段值都插入相应的 `Entry` 组件。记录 ID 必须转换成字符串，否则无法显示。

`updateAddress` 方法（第 148~183 行）更新一条现有的数据库记录。如果试图更新并不存在的记录，程序将显示一条错误消息。第 151 行检测当前记录的 id 是否有效。第 155~162 行创建 SQL `UPDATE` 语句。第 165~177 行连接并更新数据库。

`clearContents` 方法（第 185~191 行）在用户单击 Clear 按钮之后清除文本字段。`help` 方法（第 193~199 页）调用一个 `tkMessageBox` 函数，显示程序的使用指南。

17.9 因特网和万维网资源

本节提供了与数据库编程有关的网上资源。

www.mysql.com

提供可免费下载的 MySQL 数据库、最新的文档以及和开放源码许可证有关的信息。

www3.one.net/~jhoffman/sqltut.html

“Introduction to SQL” 提供一个 SQL 教程、SQL 语言资源链接以及一些例子。

www.python.org/topics/databases

可通过这里的链接访问 `MySQLdb` 等模块链接、文档、数据库编程参考书列表以及 DB-API 规范。

www.chordate.com/gadfly.html

Gadfly 是一个免费的关系数据库，完全用 Python 写成。可在此下载数据库并查看文档。

第 18 章 进 程 管 理

学习目标

- 理解进程记号法
- 理解如何创建和管理进程
- 学习如何在 Python 中执行 shell 命令
- 理解如何控制进程的输入和输出
- 学习发送和解释信号

18.1 概述

人的身体能“并行”或“并发”执行大量操作。例如，呼吸、血液循环以及食物消化可同时进行。类似地，所有官能（视觉、触觉、嗅觉、味觉和听觉）可同时发挥作用。计算机也能并发地执行操作。例如，一台桌面个人电脑可以同时编译程序、向打印机发送文件以及通过网络接收电子邮件。

有两种主要方式来实现并发性。一种方式是让每个“任务”或“进程”在单独的内存空间中工作，每个都有自己的工作内存区域。不过，虽然进程可在单独的内存空间中执行，但除非这些进程在单独的处理器上执行，否则实际并不是“同时”运行的。在单处理器系统上，操作系统使用“时间分片”或“分时”技术，将处理器时间分配给许多进程。操作系统要将一小段执行时间（即“时间片”，*quantum*）分配给一个进程。进程只能执行这一小段时间，操作系统随后将另一小段时间分配给另一个进程。操作系统要通过“背景切换”，将第一个进程及其依赖的数据转移到内存中，再将新进程及其依赖的数据转移到处理器中。在多处理器系统上，每个进程都可在单独的处理器中运行，从而实现真正的“并发性”。

实现并发性的另一个办法是在程序中指定多个“执行线程”，让它们在相同的内存空间中工作。这称为“多线程处理”，即第 19 章要讲解的主题。线程比进程更有效，因为操作系统不必为每个线程创建单独的内存空间。

操作系统提供了“shell”，即代表用户执行系统命令的程序。UNIX 用户可能熟悉 Bourne shell (`sh`) 或者 C shell (`csh`) 这样的 shell，它们提供了用于执行命令的命令行接口。Windows 用户则可能熟悉 Windows 资源管理器，它是一种图形化 shell；以及 MS-DOS 提示符，它提供了命令行接口。Macintosh OS 用户可能熟悉 Finder，这也是一种图形化 shell。

有的操作系统（例如 UNIX 和 Mac OS X）提供了内建的系统命令，允许程序员创建和管理进程。POSIX（Portable Operating System Interface for UNIX）标准为 UNIX 操作系统定义了这些系统命令。大多数 UNIX 操作系统都实现了 POSIX 标准，所以提供了同一系列的函数，以便以程序化的方式创建新进程。Python 使用这些基本操作系统命令实现进程管理。

移植性提示 18.1 并不是所有操作系统都能从一个正在运行的程序创建单独的进程。所以，进程管理是移植性最差的一项 Python 特性。

18.2 os.fork 函数

在并行执行的多个任务的应用程序中，创建新进程非常有用。例如，Apache Web 服务器（版本 2.0 之前）使用多个进程处理多个客户请求。每个进程都是主 Apache 进程的一个完全相同的拷贝。在这种情况下，有效的做法是生成主 Apache 进程的完全相同的拷贝，因为每个进程都执行相同的任务（即为客户提供网页）。

新建进程的另一个办法是使用 `os.fork` 函数，它只在相容于 POSIX 的系统上可用（包括大多数版本

的 UNIX 和 Linux)。在 Windows 版本的 Python 中, `os` 模块没有定义 `os.fork` 函数, 因为 Windows 不支持用 `fork` 新建进程。相反, Windows 程序员要用多线程编程技术来完成并发任务。

常见编程错误 18.1 Python 程序在 Windows 系统上调用 `os.fork` 会造成 `AttributeError` 异常, 因为用于 Windows 的 `os` 模块没有定义 `fork` 函数。

图 18.1 描述了 `os.fork` 函数如何新建进程。程序每次执行时, 操作系统都会创建一个新进程来运行程序指令 (步骤 1)。进程还可调用 `os.fork`, 要求操作系统新建一个进程。“父进程”是调用 `os.fork` 的进程。父进程所分支 (或创建) 的任何进程都是“子进程”。每个进程都有一个不重复的“进程 ID 号”, 或称“`pid`”, 它对进程进行标识。进程调用 `fork` 函数时, 操作系统会新建一个子进程, 它本质上与父 (原始) 进程完全相同 (步骤 2)。子进程从父进程继承了多个值的拷贝, 比如全局变量和环境变量。两个进程惟一的差别就是 `fork` 的返回值: `child` (子) 进程接收返回值 0, 而父进程接收子进程的 `pid` 作为返回值。调用 `fork` 函数后, 两个进程并发执行同一个程序, 首先执行的是调用了 `fork` 之后的下一行代码。父进程和子进程既并发执行, 又相互独立; 也就是说, 它们是“异步”执行的。图 18.2 展示了 `os.fork` 函数的一个例子。

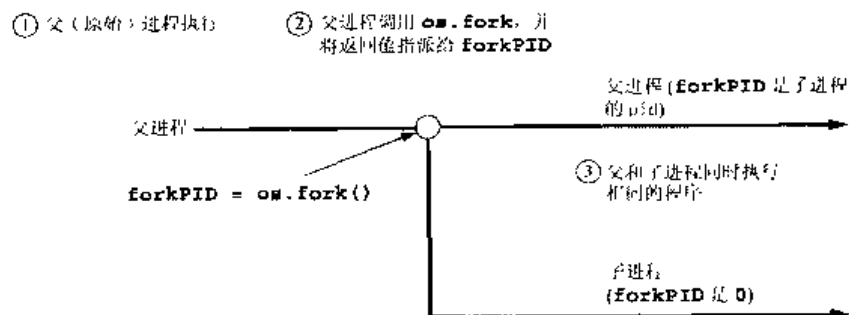


图 18.1 `os.fork` 创建一个新进程

```

1 # Fig. 18.2: fig18_02.py
2 # Using fork to create child processes.
3
4 import os
5 import sys
6 import time
7
8 processName = "parent" # only the parent is running now
9
10 print "Program executing\n\tpid: %d, processName: %s" \
11     % ( os.getpid(), processName )
12
13 # attempt to fork child process
14 try:
15     forkPID = os.fork() # create child process
16 except OSError:
17     sys.exit( "Unable to create new process." )
18
19 if forkPID != 0: # am I parent process?
20     print "Parent executing\n" + \
21         "\tpid: %d, forkPID: %d, processName: %s" \
22         % ( os.getpid(), forkPID, processName )
23
24 elif forkPID == 0: # am I child process?
25     processName = "child"
26     print "Child executing\n" + \
27         "\tpid: %d, forkPID: %d, processName: %s" \
28         % ( os.getpid(), forkPID, processName )
29
30 print "Process finishing\n\tpid: %d, processName: %s" \
31     % ( os.getpid(), processName )

```



```

Program executing
  pid: 5428, processName: parent
Parent executing
  pid: 5428, forkPID: 5429, processName: parent
Process finishing
  pid: 5428, processName: parent
Child executing
  pid: 5429, forkPID: 0, processName: child
Process finishing
  pid: 5429, processName: child

```

```

Program executing
  pid: 5430, processName: parent
Child executing
  pid: 5431, forkPID: 0, processName: child
Process finishing
  pid: 5431, processName: child
Parent executing
  pid: 5430, forkPID: 5431, processName: parent
Process finishing
  pid: 5430, processName: parent

```

```

Program executing
  pid: 5888, processName: parent
Child executing
Parent executing
  pid: 5888, forkPID: 5889, processName: parent
Process finishing
  pid: 5888, processName: parent
  pid: 5889, forkPID: 0, processName: child
Process finishing
  pid: 5889, processName: child

```

图 18.2 用 os.fork 创建子进程

移植性提示 18.2 os.fork 函数不适用于 Windows 版本的 Python。

第 8 行将变量 processName 初始化成"parent", 将当前进程指定为父进程。第 10~11 行打印父进程的 pid 和 processName。然后, 第 15 行调用 os.fork 函数, 创建当前进程的一个副本。如果操作系统不能新建进程, os.fork 函数会引发 OSError 异常, 第 17 行将退出程序; 否则, 操作系统会创建一个新进程。进程的两个副本(父与子)会从子进程创建的位置继续执行(第 15 行), 但各自位于单独的内存空间。

如果程序必须在父进程和子进程中并行执行不同任务, 程序可用 if 语句检测 fork 在每个进程中返回的值。然后, 进程可根据那些 if 语句的结果执行恰当的任务。记住 fork 在新建的子进程中返回 0; 而在父进程中, fork 返回子进程的 pid, 这必须是一个正整数。在图 18.2 的例子中, 父进程执行的任务与子进程不同。如果执行进程是父进程, fork 的返回值是子进程的 pid, 而且第 19 行的条件求值为真。然后, 进程执行父进程特有的代码(第 20~22 行)。如果执行进程是子进程, forkPID 就是 0, 第 19 行的条件求值为假。这会禁止子进程执行父进程特有的代码。相反, 第 24 行的条件求值为真, 所以执行子进程特有的代码(第 25~28 行)。

子进程把变量 processName 的副本变成值"child"(第 25 行)。程序在一个进程中修改变量值, 其他进程中的变量值不会改变。相反, 每个进程都单独包含一个名为 processName 的变量。函数 os.getpid(第 22 和第 28 行)返回当前正在执行的线程的 pid。注意, 在示范输出中, 子进程的 pid 与父进程中的 forkPID 相匹配。父进程可使用子进程的 pid 来管理子进程, 详情参见 18.7 节的说明。

注意图 18.2 的前两个示范输出是有区别的。调用 os.fork 而创建了子进程之后(第 15 行), 父进程和子进程作为异步的并发进程而单独执行。“异步”是指它们各行其是, 相互间不进行同步。“并发”是指它们可并行发生, 也就是同时执行。所以, 我们无法预测子进程和父进程的相对速度。所以, 图 18.2 的输出在每次执行时都会发生变化。有时, 父进程会在子进程执行第 26 行之前执行第 20 行; 有时却是子进程先执行。注意在最后一个示范输出中, 父进程执行第 20 行, 而子进程执行第 26 行! 这正是进程

“并发性”的本质。

输出结果不同的另一个原因是每次运行一个新进程时，操作系统都会为它指派独一无二的 pid。所以，每个进程（父与子）的 pid 在程序每次执行时都有所区别。

有时，父进程必须等子进程结束，才可继续执行。例如，子进程可能执行一个计算，而父进程需要得到计算结果才能继续。os.wait 函数（只适用于 UNIX 兼容系统）可等候子进程，并在它执行完毕之后才继续执行父进程。该函数返回包含两个元素的一个元组，包括已完成的子进程的 pid，以及子进程的“退出状态”（指出子进程退出状态的一个整数）。如返回状态为 0，表明子进程成功完成；正整数表明子进程终止时出错。如果没有子进程，os.wait 函数会引发 OSError 异常。图 18.3 演示了 os.wait 函数。

```

1 # Fig. 18.3: fig18_03.py
2 # Demonstrates the wait function.
3
4 import os
5 import sys
6 import time
7 import random
8
9 # generate random sleep times for child processes
10 sleepTime1 = random.randrange( 1, 6 )
11 sleepTime2 = random.randrange( 1, 6 )
12
13 # parent ready to fork first child process
14 try:
15     forkPID1 = os.fork() # create first child process
16 except OSError:
17     sys.exit( "Unable to create first child. " )
18
19 if forkPID1 != 0: # am I parent process?
20
21     # parent ready to fork second child process
22     try:
23         forkPID2 = os.fork() # create second child process
24     except OSError:
25         sys.exit( "Unable to create second child." )
26
27     if forkPID2 != 0: # am I parent process?
28         print "Parent waiting for child processes...\n" + \
29             "\t\t\t pid: %d, forkPID1: %d, forkPID2: %d" \
30             % ( os.getpid(), forkPID1, forkPID2 )
31
32         # wait for any child process
33         try:
34             child1 = os.wait()[ 0 ] # wait returns one child's pid
35         except OSError:
36             sys.exit( "No more child processes." )
37
38         print "Parent: Child %d finished first, one child left." \
39             % child1
40
41         # wait for another child process
42         try:
43             child2 = os.wait()[ 0 ] # wait returns other child's pid
44         except OSError:
45             sys.exit( "No more child processes." )
46
47         print "Parent: Child %d finished second, no children left." \
48             % child2
49
50     elif forkPID2 == 0: # am I second child process?
51         print ""Child2 sleeping for %d seconds...
52         "\t\t\t pid: %d, forkPID1: %d, forkPID2: %d" \
53         % ( sleepTime2, os.getpid(), forkPID1, forkPID2 )
54         time.sleep( sleepTime2 ) # sleep to simulate some work
55
56 elif forkPID1 == 0: # am I first child process?
57     print ""Child1 sleeping for %d seconds...
```

```

58      \tpid: %d, forkPID1: %d"" \
59      % ( sleepTime1, os.getpid(), forkPID1 )
60      time.sleep( sleepTime1 ) # sleep to simulate some work

```

```

Child2 sleeping for 4 seconds...
pid: 9578, forkPID1: 9577, forkPID2: 0
Child1 sleeping for 5 seconds...
pid: 9577, forkPID1: 0
Parent waiting for child processes...
pid: 9576, forkPID1: 9577, forkPID2: 9578
Parent: Child 9578 finished first, one child left.
Parent: Child 9577 finished second, no children left.

```

```

Parent waiting for child processes...
pid: 9579, forkPID1: 9580, forkPID2: 9581
Child1 sleeping for 1 seconds...
pid: 9580, forkPID1: 0
Child2 sleeping for 5 seconds...
pid: 9581, forkPID1: 9580, forkPID2: 0
Parent: Child 9580 finished first, one child left.
Parent: Child 9581 finished second, no children left.

```

```

Parent waiting for child processes...
Child1 sleeping for 4 seconds...
pid: 9583, forkPID1: 0
Child2 sleeping for 3 seconds...
pid: 9584, forkPID1: 9583, forkPID2: 0
pid: 9582, forkPID1: 9583, forkPID2: 9584
Parent: Child 9584 finished first, one child left.
Parent: Child 9583 finished second, no children left.

```

图 18.3 用 os.wait 等候子进程

该程序创建两个子进程，父进程在两个子进程都执行完毕之后才会终止。每个子进程都调用 `time.sleep` 函数，以“休眠”随机秒数（具体在第 10~11 行计算）。调用 `sleep` 函数的目的是使子进程看起来在执行一些任务。

第 15 行创建第一个子进程，第 23 行创建第二个。外层 `if` 语句（第 19 行）对 `forkPID1` 进行求值，它是第一个 `fork` 调用（第 14 行）的返回值。如父进程正在运行，就执行第 21~48 行的代码。父进程新建一个子进程（第 23 行），将返回值指派给 `forkPID2`，并检查这个变量的值，判断父进程是否仍在运行（第 27 行）。

创建了第二个子进程之后，父进程打印一条消息，指出父进程将等候它的子进程（第 28~30 行）。然后，父进程调用 `os.wait` 函数（第 34 行）。父进程会一直等候，直到它的一个子进程结束执行。注意，对于父创建的第一个子进程不必这样做。由于并发性的本质（异步进程），第二个进程有可能先执行完毕。如果父进程必须使用子进程的结果，否则就无法继续（例如父进程发送电子邮件，其中包含由一个子进程提供的邮件正文），碰到这种情况，最好用 `os.wait` 函数。

第一个子进程（第 57~60 行）打印它的 `pid` 以及 `forkPID1` 的值。然后调用 `time.sleep` 函数（第 60 行），并用一个参数指定时间长度（以秒为单位），从而规定进程应该保持“休眠”的时间。我们传递一个随机值（`sleepTime1`），模拟执行某些工作。尽管程序员不能依赖 `time.sleep` 函数进行同步，但在运行程序时，注意，父进程确实会等待正确的秒数，然后终于执行。第一个子进程唤醒后，它会结束执行，并成功终止。第二个子进程执行类似的任务（第 51~54 行）。该进程打印一条消息，并在休眠随机时间后终止。

`os.wait` 函数要求父进程等候它的任何一个子进程结束执行。然后，`os.wait` 函数返回已结束的子进程的 `pid`，而父进程恢复执行。一个子进程终止后，父进程会被唤醒，并在屏幕上打印消息，显示已经结束的子进程的 `pid`（第 38~39 行）。记住，进程既是并发的，也是异步的，所以无法预测哪个子进程先结束。第 43 行再次调用 `os.wait` 函数，这使父进程再次暂停执行，直到另一个子进程终止。另一个子进程终止时，父进程会打印一条消息，显示那个子进程的 `pid`（第 47~48 行）。

注意，每次执行时，进程的执行顺序是有区别的（参见图 18.3 的 3 个示范输出）。在第一个输出中，第二个子进程（pid 9578）首先结束。这再次提醒我们注意进程执行的并发及异步本质。

要指示父进程等候一个指定的子进程终止，可在父进程中使用 `os.waitpid` 函数。该函数只适用于 UNIX 兼容系统，它可等候一个指定的进程结束，然后返回一个双元素元组，其中包含子进程的 pid 以及那个进程的退出状态。如指定的 pid 不存在，或者子进程在调用 `waitpid` 之前便已终止，`waitpid` 函数就会引发 `OSError` 异常。函数调用将 pid 作为第一个参数传递，并将一个“选项”作为第二个参数传递。如第一个参数大于 0，`waitpid` 会等候具有指定 pid 的进程。如第一个参数是 -1，`waitpid` 会等候当前进程的任何子进程，也就是具有和 `os.wait` 函数一样的行为。如果是正常操作，传给 `os.waitpid` 的“选项”参数应该是 0。如将该参数设为常量 `os.WNOHANG`，那么对于具有指定 pid 的进程，如果没有可用的状态信息，`waitpid` 函数调用就应立即返回。

图 18.4 演示 `os.waitpid` 的用法。该程序类似于图 18.3，但其中的第 29 行将第二个子进程的 pid 以及“选项”值 0 传给 `waitpid`。这意味着父进程在第二个子进程（具有 pid 值 `forkPID2` 的进程）终止之后，才会执行第 33~34 行（不管第一个子进程是否终止）。该程序的 3 个示范输出再次证明了异步并发进程不可预测的本质。

```

1 # Fig. 18.4: fig18_04.py
2 # Demonstrates the waitpid function.
3
4 import os
5 import sys
6 import time
7
8 # parent about to fork first child process
9 try:
10     forkPID1 = os.fork() # create first child process
11 except OSError:
12     sys.exit( "Unable to create first child. " )
13
14 if forkPID1 != 0: # am I parent process?
15
16     # parent about to fork second child process
17     try:
18         forkPID2 = os.fork() # create second child process
19     except OSError:
20         sys.exit( "Unable to create second child." )
21
22     if forkPID2 > 0: # am I parent process?
23         print "Parent waiting for child processes...\n" + \
24             "\t pid: %d, forkPID1: %d, forkPID2: %d" \
25             % ( os.getpid(), forkPID1, forkPID2 )
26
27         # wait for second child process explicitly
28         try:
29             child2 = os.waitpid( forkPID2, 0 )[ 0 ] # child's pid
30         except OSError:
31             sys.exit( "No child process with pid %d." % ( forkPID2 ) )
32
33         print "Parent: Child %d finished." \
34             % child2
35
36     elif forkPID2 == 0: # am I second child process?
37         print "Child2 sleeping for 4 seconds...\n" + \
38             "\t pid: %d, forkPID1: %d, forkPID2: %d" \
39             % ( os.getpid(), forkPID1, forkPID2 )
40         time.sleep( 4 )
41
42     elif forkPID1 == 0: # am I first child process?
43         print "Child1 sleeping for 2 seconds...\n" + \
44             "\t pid: %d, forkPID1: %d" % ( os.getpid(), forkPID1 )
45         time.sleep( 2 )

```



```

Parent waiting for child processes...
  pid: 6092, forkPID1: 6093, forkPID2: 6094
Child1 sleeping for 2 seconds...
  pid: 6093, forkPID1: 0
Child2 sleeping for 4 seconds...
  pid: 6094, forkPID1: 6093, forkPID2: 0
Parent: Child 6094 finished.

```

```

Child1 sleeping for 2 seconds...
  pid: 6089, forkPID: 0
Child2 sleeping for 4 seconds...
  pid: 6090, forkPID: 6089, forkPID2: 0
Parent waiting for child processes...
  pid: 6088, forkPID: 6089, forkPID2: 6090
Parent: Child 6090 finished.

```

```

Parent waiting for child processes...
Child1 sleeping for 2 seconds...
  pid: 6102, forkPID: 0
  pid: 6101, forkPID: 6102, forkPID2: 6103
Child2 sleeping for 4 seconds...
  pid: 6103, forkPID: 6102, forkPID2: 0
Parent: Child 6103 finished.

```

图 18.4 用于等候特定子进程的 os.waitpid

18.3 os.system 函数和 os.exec 函数家族

Python 提供几种方式从 Python 代码中执行系统命令和其他程序。一种方式是调用 os.system 函数，它使用 shell 来执行系统命令，然后在命令结束之后将控制权返回给原始进程。另一种方式是通过 os.exec 这个“函数家族”来执行程序。和 os.system 不同，所有 os.exec 函数在执行了指定命令之后，都不将控制权返回给调用进程。程序调用 os.exec 函数时，由 os.exec 函数执行的程序会“接管”Python 进程。事实上，Python 进程会在调用 os.exec 函数时立即终止。然后执行被调用的程序，它的 pid 与之前执行的 Python 进程一样。os.system 函数和 os.exec 函数家族适用于 UNIX 和 Windows 系统。本节要创建实际程序，根据用户的操作系统来采取系统特有的行动。

图 18.5 使用 os.system 函数创建了一个简单的字处理程序。它请求用户输入要创建的文件的名称。然后，由用户输入文件内容。用户可输入 clear 以删除文件内容，或输入 quit 以保存文件并退出程序。图 18.6 显示了用这个程序创建的一个文件的内容。图 18.5 中，前两个示范输出分别是 Windows 系统和 UNIX 兼容系统上的结果，其余输出在两种系统上都是一样的。

```

1 # Fig. 18.5: fig18_05.py
2 # Uses the system function to clear the screen.
3
4 import os
5 import sys
6
7 def printInstructions( clearCommand ):
8     os.system( clearCommand ) # clear display
9
10    print """Type the text that you wish to save in this file.
11    Type clear on a blank line to delete the contents of the file.
12    Type quit on a blank line when you are finished.\n"""
13
14    # determine operating system
15    if os.name == "nt" or os.name == "dos": # Windows system
16        clearCommand = "cls"
17        print "You are using a Windows system."
18    elif os.name == "posix": # UNIX-compatible system
19        clearCommand = "clear"
20        print "You are using a UNIX-compatible system."

```

```

21 else:
22     sys.exit( "Unsupported OS" )
23
24 filename = raw_input( "What file would you like to create? " )
25
26 # open file
27 try:
28     file = open( filename, "w+" )
29 except IOError, message:
30     sys.exit( "Error creating file: %s" % message )
31
32 printInstructions( clearCommand )
33 currentLine = ""
34
35 # write input to file
36 while currentLine != "quit\n":
37     file.write( currentLine )
38     currentLine = sys.stdin.readline()
39
40     if currentLine == "clear\n":
41
42         # seek to beginning and truncate file
43         file.seek( 0, 0 )
44         file.truncate()
45
46         currentLine = ""
47         printInstructions( clearCommand )
48
49 file.close()

```

You are using a Windows system.
What file would you like to create? welcome.txt

You are using a UNIX-compatible system.
What file would you like to create? welcome.txt

Type the text that you wish to save in this file.
Type clear on a blank line to delete the contents of the file.
Type quit on a blank line when you are finished.

This will not be written to the file.
The following line will call clear.
clear

Type the text that you wish to save in this file.
Type clear on a blank line to delete the contents of the file.
Type quit on a blank line when you are finished.

Type the text that you wish to save in this file.
Type clear on a blank line to delete the contents of the file.
Type quit on a blank line when you are finished.

Willkommen!
Bienvenue!
Welcome!
quit

图 18.5 在简单字处理程序中使用 os.system

Willkommen!
Bienvenue!
Welcome!

图 18.6 Windows 或 UNIX 系统的 welcome.txt 文件的内容

不同操作系统用不同命令从控制台清除文本。第 15~22 行通过识别用户的操作系统，并相应地设置 clearCommand 命令，使程序具有更好的移植性。os.name 变量包含了用户操作系统的名称。如果该名

称是"nt"或"dos",表明使用的是 Windows 或 MS-DOS 操作系统,所以将 `clearCommand` 变量设为"cls",它可在此类操作系统中清除控制台(第16行)。如果该名称是"posix",表明程序正在实现了 POSIX 标准的操作系统上运行(比如大多数版本的 UNIX)。因此,`clearCommand` 变量要设为"clear"(第19行)。如果名称与上述任何字符串都不匹配,程序会退出,并显示错误消息(第22行)。

确定操作系统类型后,程序提示输入文件名(第24行),并试图创建该文件(第27~30行)。第32行调用 `printInstructions` 函数(在第7~12行定义)来清除屏幕显示,并打印操作指示。第8行调用 `os.system` 函数,执行系统特有的 shell 命令来清除控制台。程序将要执行的 shell 命令作为一个字符串参数传给该函数,然后等待 shell 命令执行。命令结束后,控制权会回到程序,第10~12行打印操作指示。

第36~47行将用户输入的行写入文件。第37行调用文件对象方法 `write`,将文本写入文件。然后,程序从用户处获得下一个行。如果该行是命令"clear",第43~44行就转移到文件头,并对其进行 `truncate` 处理,删除文件的全部内容。应用程序还调用 `printInstructions` 函数以清除控制台,并重新打印操作提示(第47行)。如果用户输入"quit",程序就会终止 `while` 循环、关闭文件(第49行)并退出。

图 18.7 显示了较复杂的文本编辑例子。它使用 `urllib` 模块 `urlretrieve` 函数来获得一个网页,并将其存储到文件中。然后,它用 `os.execvp` 函数启动一个外部编辑软件来编辑这个页。`os.execvp` 函数会将 Python 进程替换成由 `os.execvp` 执行的程序。控制权转交给另一个进程后,如果原来的进程希望终止,就必须进行这样的操作。注意,在运行程序时,Python 程序不会像以前那样等待,而是在编辑器启动之后立即终止。第一个示范输出显示了用于调用程序的命令。最后两个示范输出则分别显示了在 UNIX 和 Windows 机器上运行该程序的结果。

在许多系统上(尤其是 DOS 和 UNIX),可采取命令行参数的形式,将信息从命令行传递给程序。Python 将命令行参数存储在一个名为 `sys.argv` 的列表中。该列表至少包含一个值,即用户在命令行输入的程序名,其他还可包含命令行参数,它们遵循的是用户将参数传给程序时的顺序。

```

1 # Fig. 18.7: fig18_07.py
2 # Opens a Web page in a system-specific editor.
3
4 import os
5 import sys
6 import urllib
7
8 if len( sys.argv ) != 3:
9     sys.exit( "Incorrect number of arguments." )
10
11 # determine operating system and set editor command
12 if os.name == "nt" or os.name == "dos":
13     editor = "notepad.exe"
14 elif os.name == "posix":
15     editor = "vi"
16 else:
17     sys.exit( "Unsupported OS" )
18
19 # obtain Web page and store in file
20 urllib.urlretrieve( sys.argv[ 1 ], sys.argv[ 2 ] )
21
22 # editor expects to receive itself as an argument
23 os.execvp( editor, ( editor, sys.argv[ 2 ] ) )
24
25 print "This line never executes."

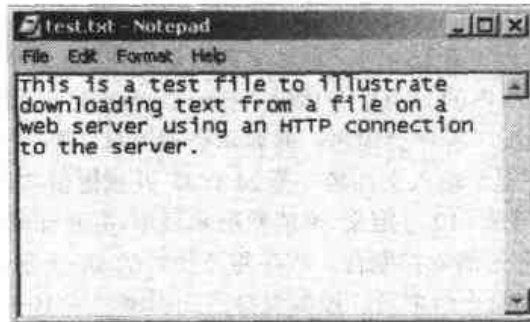
```

```
fig18_07.py http://www.deitel.com/test/test.txt test.txt
```

```

This is a test file to illustrate
downloading text from a file on a
web server using an HTTP connection
to the server.
~
~
~

```

图 18.7 用 `os.execvp` 将控制权彻底移交给另一个程序

在我们的例子中，程序要求用户将两个命令行参数传给程序：要访问的 URL，以及用于保存 URL 内容的一个文件的名称。所以，`sys.argv` 列表的长度应为 3（程序名加两个参数）。第 8~9 行检测这个值，并判断用户是否传递了数目正确的参数。

第 13 和第 15 行指定编辑器命令，这要取决于 `os.name` 变量中存储的操作系统类型。如操作系统是 Windows，应用程序就使用记事本程序；如操作系统实现了 POSIX 标准，就使用 vi 编辑器。如果不是这两种操作系统，程序显示错误消息并退出。注意，在所有操作系统上，`PATH` 环境变量必须包括到编辑器命令的路径。例如对于记事本程序（`notepad.exe`），在 Windows XP 上的路径是 `C:\WINDOWS`；在 Windows NT/2000 上是 `C:\WINNT`。在大多数 UNIX 系统上，vi 的路径是 `/bin`。

第 20 行调用 `urllib.urlretrieve` 函数以获得网页。传给它的第一个参数是网页 URL（指定为字符串），第二个参数是文件名，获取的数据要写入该文件。

第 23 行调用 `os.execvp` 函数，将当前进程替换成本文本编辑程序。该函数是带有 `exec` 前缀的“函数家族”的一部分。`execvp` 函数取得两个参数。第一个是要执行的命令名，第二个是由传给命令的参数所构成的一个元组。该元组采取的是一种特殊格式：元组的第一个元素是命令本身，后续元素分别对应于各个命令行参数。在我们的例子中，传递的是新建文件的名称。图 18.8 总结了 `exec` 家族的各个函数。

函数名	说明
<code>exec1(path, arg0, arg1, ...)</code>	用指定的参数执行程序
<code>execle(path, arg0, arg1, ..., environment)</code>	与 <code>exec1</code> 相同，但取得一个附加的参数，即程序的执行环境。该环境要以一个字典的形式指定，其中的“键”是环境变量，“值”是环境变量的值
<code>execlp(path, arg0, arg1, ...)</code>	与 <code>exec1</code> 相同，只是要在 <code>os.environ["PATH"]</code> 中搜索可执行程序的路径（ <code>path</code> ）
<code>execv(path, arguments)</code>	与 <code>exec1</code> 相同，但参数包含在单个元组或列表中
<code>excve(path, arguments, environment)</code>	与 <code>execle</code> 相同，但参数包含在单个元组或列表中
<code>execvp(path, arguments)</code>	与 <code>execlp</code> 相同，但参数包含在单个元组或列表中
<code>execvpe(path, arguments, environment)</code>	与 <code>execvp</code> 相同，但要取得附加的 <code>environment</code> 参数，指定程序的执行环境

图 18.8 `exec` 函数家族

18.4 控制进程的输入和输出

UNIX 系统的基本思路就是程序应尽可能小，而且每个都要执行良好定义的任务。然后，用户可同时使用多个程序来获得所需的结果。例如，UNIX 命令 `ls` 用于列出目录内容。UNIX 命令 `sort` 根据一些排序方案（例如字母顺序）对它的输入进行排序。以下是同时使用 3 个命令的一个例子，它可生成目录

内容的一个排序列表：

```
ls | sort
```

上述命令使用管道符(|)将来自ls的输出(stdout)“管道化”(传输)到sort命令的输入(stdin)。ls命令列出目录内容，sort命令按字母顺序对内容排序，生成排好序的目录列表。

Python 提供了专门的函数来访问 stdout、stdin 和 stderr 流(在 14.3 节定义)的内容。通过使用这些函数，程序就能处理其他程序(比如以上命令行)的输入和输出。这些函数返回一个或多个文件对象，它们对应于程序的输入和输出流。处理这些文件对象时，具体方式和处理其他任何文件对象无异。要将来自一个程序的信息变成另一个程序的输入，第一个程序要打印到或者写入到第二个程序的输入文件对象——使数据传输到第二个程序的输入。要从第二个程序读取输出，第一个程序要从第二个程序的输出文件对象读取。

图 18.9 显示了一个例子，它可逆序生成目录列表，具体方式是取得一个命令的输出，并将输出内容送给另一个命令作为输入。之后，Python 程序读取并显示来自第二个命令的输出。示范输出分别显示了程序在 UNIX 和 Windows 机器上运行的结果。

```
1 # Fig. 18.9: fig18_09.py
2 # Demonstrating popen and popen2.
3
4 import os
5
6 # determine operating system, then set directory-listing and
7 # reverse-sort commands
8 if os.name == "nt" or os.name == "dos": # Windows system
9     fileList = "dir /B"
10    sortReverse = "sort /R"
11 elif os.name == "posix": # UNIX-compatible system
12     fileList = "ls -l"
13     sortReverse = "sort -r"
14 else:
15     sys.exit("OS not supported by this program.")
16
17 # obtain stdout of directory-listing command
18 dirOut = os.popen( fileList, "r" )
19
20 # obtain stdin, stdout of reverse-sort command
21 sortIn, sortOut = os.popen2( sortReverse )
22
23 filenames = dirOut.read() # output from directory-listing command
24
25 # display output from directory-listing command
26 print "Before sending to sort"
27 print "(Output from '%s'):" % fileList
28 print filenames
29
30 sortIn.write( filenames ) # send to stdin of sort command
31
32 dirOut.close() # close stdout of directory-listing command
33 sortIn.close() # close stdin of sort command -- sends EOF
34
35 # display output from sort command
36 print "After sending to sort"
37 print "(Output from '%s'):" % sortReverse
38 print sortOut.read() # output from sort command
39
40 sortOut.close() # close stdout of sort command
```

```
Before sending to sort
(Output from 'ls -l'):
fig18_02.py
fig18_03.py
fig18_04.py
fig18_05.py
fig18_07.py
```

```

fig18_09.py
fig18_10.py
fig18_14.py
fig18_15.py

After sending to sort
(Output from 'sort -r'):
fig18_15.py
fig18_14.py
fig18_10.py
fig18_09.py
fig18_07.py
fig18_05.py
fig18_04.py
fig18_03.py
fig18_02.py

Before sending to sort
(Output from 'dir /B'):
fig18_02.py
fig18_03.py
fig18_04.py
fig18_05.py
fig18_07.py
fig18_09.py
fig18_10.py
fig18_14.py
fig18_15.py

After sending to sort
(Output from 'sort /R'):
fig18_15.py
fig18_14.py
fig18_10.py
fig18_09.py
fig18_07.py
fig18_05.py
fig18_04.py
fig18_03.py
fig18_02.py

```

图 18.9 用于连接两个进程的 os.popen/os.popen2

程序调用 shell 命令列出当前目录的内容，然后将目录列表传给另一个 shell 命令，以便按照与字母表相反的顺序排列各个项目。第 8~15 行判断用户的操作系统，并相应地设置两个命令。

第 18 行调用 os.popen 函数以执行命令，并获得命令的 stdout 流（dirOut）。函数要取得两个参数，一个是要执行的 shell 命令，另一个是调用函数时所用的“模式”。如果模式为“r”（代表“读取”），表明函数返回的文件对象对应于 shell 命令的 stdout；模式为“w”（代表“写入”），表明要为命令的 stdin 返回一个文件对象。

第 21 行调用 os.popen2 函数以执行排序命令，并获得命令的 stdout 流和 stdin 流。函数返回一个元组，其中含有两个文件对象。第一个对象是 sortIn，它对应于命令的 stdin 流；第二个对象是 sortOut，对应于命令的 stdout 流。

接着，程序将第一个命令的输出送给第二个命令作为输入，以便对目录列表排序。第 23 行调用 read 方法，获得目录命令的输出。第 28 行打印这个输出。然后，第 30 行使用文件方法 write，将输出传给排序命令的 stdin。这样便把第一个命令的输出与第二个命令的输入联系在一起。然后，程序关闭两个文件对象（第 32~33 行）。要想向排序命令发送 EOF（文件尾）字符必须关闭与排序命令的 stdin 对象对应的文件对象。排序命令只有在接收到这个字符后，才会开始对输入排序。

关闭排序命令的 stdin 对象后，命令会按逆序对目录列表排序。我们可读取命令的输出（第 38 行）。程序结束时，会关闭排序命令的 stdout 对象（第 40 行）。

18.5 进程间通信

os 模块为多种“进程间通信”(InterProcess Communication, IPC) 机制提供了一个接口。进程使用 IPC 机制在进程间传递信息。一种 IPC 机制是“管道”，它是一种类似于文件的对象，提供单向通信渠道。父进程可打开一个管道，再分支(创建)一个子进程。父进程使用管道将信息写入(发送到)子进程，而子进程使用管道从父进程读取信息。

在 Python 中用 os.pipe 函数创建管道。该函数返回一个元组，其中包含两个文件说明符。“文件说明符”是一个数字，操作系统用它来表示一个打开的文件。第一个文件说明符提供了对管道的“读”访问；第二个文件提供了对管道的“写”访问。文件说明符和 Python 文件对象的区别在于，文件对象封装了一个文件说明符，并提供了对文件内容进行修改的方法。可调用 os.read 和 os.write 方法读写与文件说明符对应的文件。图 18.10 的程序使用管道在父进程和子进程之间进行通信。

```

1 # Fig. 18.10: fig18_10.py
2 # Using os.pipe to communicate with a child process.
3
4 import os
5 import sys
6
7 # open parent and child read/write pipes
8 fromParent, toChild = os.pipe()
9 fromChild, toParent = os.pipe()
10
11 # parent about to fork child process
12 try:
13     pid = os.fork() # create child process
14 except OSError:
15     sys.exit( "Unable to create child process." )
16
17 if pid != 0: # am I parent process?
18
19     # close unnecessary pipe ends
20     os.close( toParent )
21     os.close( fromParent )
22
23     # write values from 1-10 to parent's write pipe and
24     # read 10 values from child's read pipe
25     for i in range( 1, 11 ):
26         os.write( toChild, str( i ) )
27         print "Parent: %d," % i,
28         print "Child: %s" % \
29             os.read( fromChild, 64 )
30
31     # close pipes
32     os.close( toChild )
33     os.close( fromChild )
34
35 elif pid == 0: # am I child process?
36
37     # close unnecessary pipe ends
38     os.close( toChild )
39     os.close( fromChild )
40
41     # read value from parent pipe
42     currentNumber = os.read( fromParent, 64 )
43
44     # if we receive number from parent,
45     # write number to child write pipe
46     while currentNumber:
47         newNumber = int( currentNumber ) * 20
48         os.write( toParent, str( newNumber ) )
49         currentNumber = os.read( fromParent, 64 )
50
51     # close pipes

```

```

52 os.close( toParent )
53 os.close( fromParent )
54 os._exit( 0 ) # terminate child process

```

```

Parent: 1, Child: 20
Parent: 2, Child: 40
Parent: 3, Child: 60
Parent: 4, Child: 80
Parent: 5, Child: 100
Parent: 6, Child: 120
Parent: 7, Child: 140
Parent: 8, Child: 160
Parent: 9, Child: 180
Parent: 10, Child: 200

```

图 18.10 用 os.pipe 在父进程和子进程之间通信

第 8~9 行调用 `os.pipe` 函数以创建管道，父进程和子进程将利用它进行通信。在这个例子中，父进程向子进程发送信息，子进程也向父进程发送信息。第 8 行创建管道，以便父进程向子进程发送信息。变量 `fromParent` 引用的是从管道读取数据的文件说明符；变量 `toChild` 引用的是向管道写入数据文件说明符。第 9 行为子进程获得类似的文件说明符。注意，管道是先于子进程创建的（第 13 行）。这样一来，父进程和子进程执行时，都能访问文件描述符。代码的父进程部分（第 19~33）行首先调用 `os.close` 函数关闭父进程不使用的两个文件说明符，即 `fromParent` 和 `toParent`（第 20~21 行）。作为一个好习惯，进程应主动关闭自己不需要的那一端的管道。父进程和子进程各自都有这些变量的副本，所以，在一个进程中关闭文件说明符不会关闭另一个进程中的文件说明符。

图 18.11~图 18.13 演示了程序如何创建两个管道，并通过它实现父进程和子进程的通信。每个进程都向自己的管道写入，并从对方的管道读入。图 18.11 显示了初始阶段：父进程创建两个管道。在这个阶段，父进程是惟一正在执行的进程，它可对两个管道进行读写。接着，父进程创建新的子进程。针对父进程中的所有变量，这个子进程都获得它自己的一个副本，其中包括两个管道（图 18.12）。但为了进行通信，父进程只需从子管道读入，以及向父管道写入；而子进程只需从父管道读入，以及向子管道写入。所以，每个进程都应关闭每个管道的自己不需要的另一端。父进程关闭的是自己管道的读入端和子管道的写入端；而子进程关闭的是自己管道的读入端和父管道的写入端。完成这些操作后，父进程和子进程就能通过合适的管道进行通信（图 18.13）。

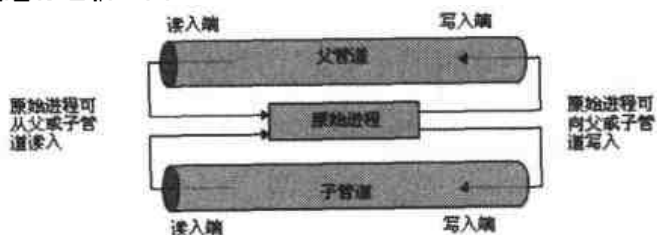


图 18.11 初始阶段：原始进程可以读写两个管道

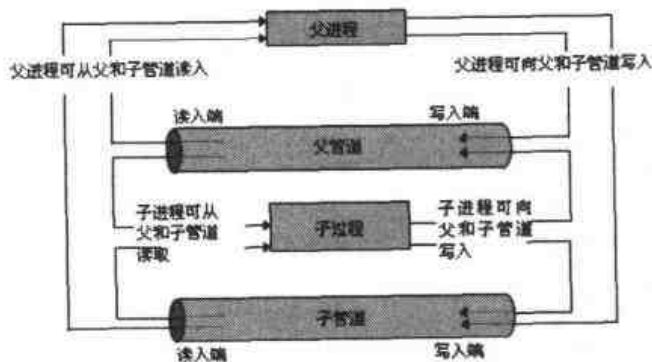


图 18.12 中间阶段：父进程和子进程可以读写两个管道

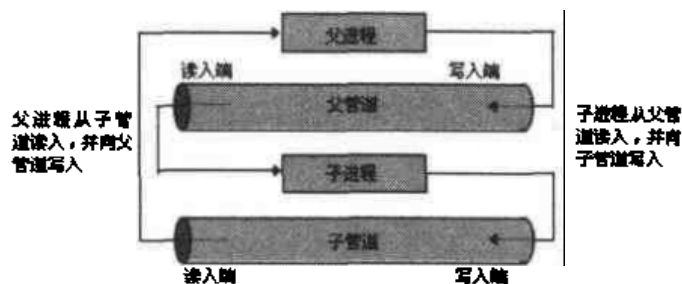


图 18.13 最后阶段：父进程和子进程关闭管道不需要的那一端

第 25~29 行将 1~10 的数字写入父管道。os.write 函数取得两个参数：一个是文件说明符，另一个是要写入与文件说明符对应的文件中的字符串。第 26 行的语句将数据插入管道，数据会暂时在管道中等待，直到子进程读取那个数据。

第 28~29 行读入并打印从子管道获得的数据。os.read 函数取得两个参数，一个是文件说明符，另一个是要读入的数据长度（以字节为单位）。该函数将使调用进程暂停，直到从管道获得数据。换言之，除非管道里包含了一些数据，否则 os.read 函数不会返回到调用函数。这会导致父进程暂停，直到子进程将数据写入管道。子进程在父管道中写入 10 个值后，父进程会关闭父管道的写入端以及子管道的读入端。

子进程（第 37~54 行）首先关闭不需要的文件说明符，即 toChild 和 fromChild（第 38~39 行）。然后，子进程使用 os.read 函数，从父管道读入一个值（第 42 行）。第 46~49 行包含一个 while 循环，它将从父管道读入的值乘以 20，然后将乘积写入子管道。然后，子进程从父进程那里读取下一个数字。如管道关闭，os.read 函数会返回空字符串，而且 while 循环退出，子进程则关闭它其余的文件说明符。

第 54 行调用函数 os._exit 以终止子进程。os._exit 类似 sys.exit，但它不执行任何清除工作（例如刷新缓冲区）。所以，os._exit 尤其适用于退出子进程。如果程序使用 sys.exit，操作系统就会回收父进程或其他子进程可能仍然需要的资源。传给 os._exit 函数的参数必须是进程的退出状态。退出状态为 0，表示正常终止。

良好编程习惯 18.1 进程应关闭不需要的管道端，因为操作系统限制了可同时打开的文件说明符数量。

18.6 信号处理

进程也可用信号进行通信。所谓“信号”，是操作系统采取异步方式传给程序的消息。例如在 Windows 或 UNIX 控制台程序执行期间按 Ctrl-C，操作系统会向程序发送一个“中断信号”。通常，该信号导致程序终止。然而，程序完全可以指定用不同的行动来响应任何一个信号。在“信号处理”中，程序要接收信号，并根据那个信号采取一项行动。错误（例如向已关闭的管道写入）、事件（例如计时器变成 0）以及用户输入（例如按 Ctrl-C）都会生成信号。除此之外，程序内部也可能生成信号。

为响应特定的信号，需要执行“信号处理程序”。针对每个信号，每个 Python 程序都有一个默认的信号处理程序。例如，假定 Python 解释器收到一个信号，该信号指出程序试图向已关闭的管道写入，或者用户敲入一个键盘中断，Python 就会引发一个异常。发生这种异常后，程序既可使用默认处理程序，也可使用自定义处理程序。

图 18.14 显示了为中断信号设置自定义处理程序的例子。在大多数平台上按 Ctrl-C 都会发送一个中断信号。这个程序覆盖了中断信号处理程序，要求用户按 3 次 Ctrl-C 才能中断程序。第 5 行导入 signal 模块，它为 Python 程序提供了信号处理能力。

```
1 # Fig. 18.14: fig18_14.py
2 # Defining our own signal handler.
3
4 import time
5 import signal
```



```

6
7 def stop( signalNumber, frame ):
8     global keepRunning
9     keepRunning -= 1
10    print "Ctrl-C pressed; keepRunning is", keepRunning
11
12    keepRunning = 3
13
14    # set the handler for SIGINT to be function stop
15    signal.signal( signal.SIGINT, stop )
16
17    while keepRunning:
18        print "Executing..."
19        time.sleep( 1 )
20
21    print "Program terminating..."

```

```

Executing...
Executing...
Ctrl-C pressed; keepRunning is 2
Executing...
Executing...
Ctrl-C pressed; keepRunning is 1
Executing...
Executing...
Ctrl-C pressed; keepRunning is 0
Program terminating...

```

图 18.14 自定义信号处理程序

移植性提示 18.3 每个系统都定义了特有的信号集。signal 是依赖于具体平台的模块，其中只包含系统定义的信号。

该示例程序首先使用 signal.signal 函数，为中断信号注册一个信号处理程序（第 15 行）。函数要获取两个参数：一个信号和一个对应于信号处理程序的函数。传给 signal.signal 的第二个参数也可以是某个标准的信号处理程序，即 signal.SIG_IGN（忽略信号）或者 signal.SIG_DFL（信号的默认处理程序）。signal.SIGINT 值代表中断信号，stop 函数名称，我们将这个函数用作信号处理程序。第 15 行要求程序在收到一个中断信号时（例如按 Ctrl-C），就调用 stop 函数。

stop 函数（第 7~10 行）实现了信号处理程序。信号处理程序要取得两个参数。第一个参数是程序收到的信号，第二个参数包含当前堆栈帧（stack frame）。下一个例子将更详细地讨论第一个参数，堆栈帧的问题已在 12.7 节简单地说明。在这个例子中，信号处理程序每次执行时，都会使全局变量 keepRunning 自减。

注册了信号处理程序之后，如果用户按 Ctrl-C，就会执行 stop 函数。程序收到一个信号时，正在执行代码会被中断，使程序能处理信号。这通常不是问题，因为信号处理完毕后，会从中断位置继续执行以前的代码。在这种情况下，我们说代码可以“重新进入”。然而，有的代码（例如 UNIX 系统调用或者用于处理内存的内部 Python 语句）不能重新进入。信号中断这种代码后，代码不能重新开始。

第 12 行将变量 keepRunning 初始化为 3。第 17 行则开始一个 while 循环：只要 keepRunning 大于 0，就一直进行这个循环。如果用户按了 3 次 Ctrl-C，信号处理程序会使 keepRunning 的值自减为 0，然后循环退出。此时，程序会在打印一条消息后终止（第 21 行）。

18.7 发送信号

上一节演示了进程如何处理信号。此外，我们还要注意，正在执行程序可将信号发送给其他进程。os.kill 函数能将信号发送给由 pid 标识的特定进程。本节要讨论一个例子（图 18.15），由父进程将信号发送给它的子进程。该示例程序在父进程中处理中断信号，其中，父进程要向子进程发送一个信号，以终止子进程。

```

1 # Fig. 18.15: fig18_15.py
2 # Sending signals to child processes using kill
3
4 import os
5 import signal
6 import time
7 import sys
8
9 # handles both SIGALRM and SIGINT signals
10 def parentInterruptHandler( signum, frame ):
11     global pid
12     global parentKeepRunning
13
14     # send kill signal to child process and exit
15     os.kill( pid, signal.SIGKILL ) # send kill signal
16     print "Interrupt received. Child process killed."
17
18     # allow parent process to terminate normally
19     parentKeepRunning = 0
20
21 # set parent's handler for SIGINT
22 signal.signal( signal.SIGINT, parentInterruptHandler )
23
24 # keep parent running until child process is killed
25 parentKeepRunning = 1
26
27 # parent ready to fork child process
28 try:
29     pid = os.fork() # create child process
30 except OSError:
31     sys.exit( "Unable to create child process." )
32
33 if pid != 0: # am I parent process?
34
35     while parentKeepRunning:
36         print "Parent running. Press Ctrl-C to terminate child."
37         time.sleep( 1 )
38
39 elif pid == 0: # am I child process?
40
41     # ignore interrupt in child process
42     signal.signal( signal.SIGINT, signal.SIG_IGN )
43
44     while 1:
45         print "Child still executing."
46         time.sleep( 1 )
47
48 print "Parent terminated child process."
49 print "Parent terminating normally."

```

```

Parent running. Press Ctrl-C to terminate child.
Child still executing.
Parent running. Press Ctrl-C to terminate child.
Child still executing.
Child still executing.
Parent running. Press Ctrl-C to terminate child.
Interrupt received. Child process killed.
Parent terminated child process.
Parent terminating normally.

```

图 18.15 用 os.kill 向其他进程发送信号

示例程序首先定义一个信号处理程序，它允许父进程处理中断信号（第 10~19 行）。第 22 行在父进程中注册信号处理程序。第 25 行初始化变量 `parentKeepRunning`，父进程用它来判断何时终止。

第 29 行新建一个子进程。第 42~46 行只在新建的子进程中执行。第 42 行为子进程中的中断信号注册 `signal.SIG_IGN` 信号处理程序。这实际会导致子进程忽略中断信号。第 44~46 行是一个无限循环，用于保持子进程运行。

第 35~37 行在父进程中执行。第 36 行向用户打印操作提示。按 Ctrl-C 时, Python 解释器会调用父进程的中断信号处理程序, 即 `parentInterruptHandler` (第 10~19) 行。第 15 行调用函数 `os.kill`, 将名为 `signal.SIGKILL` 的 kill 信号发送给子进程。传给 `os.kill` 的第一个参数是进程的 pid (程序要将信号发送给该进程)。第二个参数指定了要发送的信号的类型。该程序发送 kill 信号, 以终止子进程。然后, 第 19 行将 `parentKeepRunning` 标记设为 0, 导致第 35~37 行的循环终止, 并允许父进程正常终止。

在 UNIX/Linux 系统中, 子进程终止后, 会保留在进程表中, 让父进程知道子进程是否正常终止。如果创建大量子进程, 但在终止后没有从进程表移除它们, 进程表便会积累越来越多的死进程, 这些进程称为“zombies”(僵尸进程)。消除僵尸进程的操作称为“reaping”, 这是通过 `os.wait` 或 `os.waitpid` 函数来实现的。

为避免僵尸进程越积越多, 请为 `SIGCHLD` 信号设置一个处理程序。子进程退出时, 会向父进程发送该信号。该处理程序应该主动调用 `os.wait` 或者 `os.waitpid` 函数, 并根据需要采取其他操作。如果子进程终止时无需执行额外的任务, 就将现成的 `signal.SIG_IGN` 注册成为 `SIGCHLD` 信号的处理程序。该处理程序要求进程忽略 `SIGCHLD` 信号, 并禁止子进程保留在进程表中。

第 19 章 多线程处理

学习目标

- 理解多线程处理的背景
- 理解多线程处理如何改进性能
- 理解如何创建、管理和销毁线程
- 理解线程生命期
- 研究几个线程同步的例子

19.1 概述

第 18 章讨论了如何使用进程（即在独立内存空间中执行的代码）来执行并发任务。本章要讨论多线程处理技术，它能在单独的进程中执行并发任务。线程是“轻量级”进程，因为相较于进程的创建和管理，操作系统通常会用较少的资源来创建和管理线程。操作系统要为新建的进程分配单独的内存位置和数据；相反，程序中的线程在相同的内存空间中执行，并共享许多相同的资源。多线程程序对内存的使用效率要优于多进程程序。

Python 和许多常规用途程序语言的区别在于，它提供了完整的多线程处理类。^①如果操作系统本身支持多线程，就可用 Python 的 `threading` 模块创建多线程应用程序。程序员可指定在一个应用程序中包含多个执行线程，而且每个线程都表明程序中的一部分要与其他线程并发执行。Python 的多线程支持为程序员赋予了强大的编程能力，这些能力是在其他许多语言所不具备的。

许多任务都可获益于多线程编程。Web 浏览器下载大文件时（比如音乐或视频剪辑），用户希望立即开始欣赏音乐或观看视频，不愿意等候下载完整的剪辑。所以，可让一个线程下载，另一个线程播放已经下载的那一部分。这样就实现了多个操作并发执行。为了避免播放时断时续，应在下载了足够的数据量之后，才开始播放线程。

性能提示 19.1 单线程程序的问题在于，要在结束费时较长的操作后，才能开始其他操作。而在多线程程序中，线程可共享一个或多个处理器，使多个任务并行执行。

尽管人的大脑可同时执行多项任务，但通常很难在平行的“思路”之间跳转。要想体会多线程难于编程和理解的特点，请做以下试验：同时打开 3 本书，都翻到第 1 页，尝试同时读这些书。在第一本上读几个字，从第二本上读几个字，再从第 3 本上读几个字。然后回到第一本书，往后接着读几个字……以此类推。这个试验能让您体会多线程处理所带来的挑战。在不同的书之间从容切换，快速地读每本书，记住在每本书中的位置，将正在阅读的书拿近一些以便看清楚，并将没有阅读的书推到一边，这往往很难做到。此外，经过一阵忙乱后，您几乎不可能领会这些书的内容。

19.2 线程状态：生命期

Python 解释器控制一个程序中的所有线程。解释器开始执行程序时，“主”线程开始执行。每个线程都可创建和启动其他线程。如果程序包含多个正在运行的线程，它们将依据指定的间隔时间（称为一个 *quantum*），依次进入和离开解释器。Python 的“全局解释器锁”（Global Interpreter Lock, GIL）保证解释器在任何时刻只运行一个线程。GIL 每次可用时，都会有单个线程包含着它。然后，线程进入解释

^① 许多语言虽然没有直接提供多线程处理能力，但仍可用于实现多线程程序。一些语言使用操作系统特有的线程处理库。以这种方式实现的程序在多个平台上执行时，一般都需要修改。

器，并在该线程的 quantum 时间段中执行它。一旦 quantum 到期，线程就离开解释器，同时释放 GIL。特定的操作会导致线程的 quantum 被截短。稍后就会讨论这些操作。

在任何时刻，线程都处于某种线程状态（图 19.1）。本节讨论了各种状态，并解释了状态之间的转移。另外还讨论了导致线程改变状态的对象和方法。

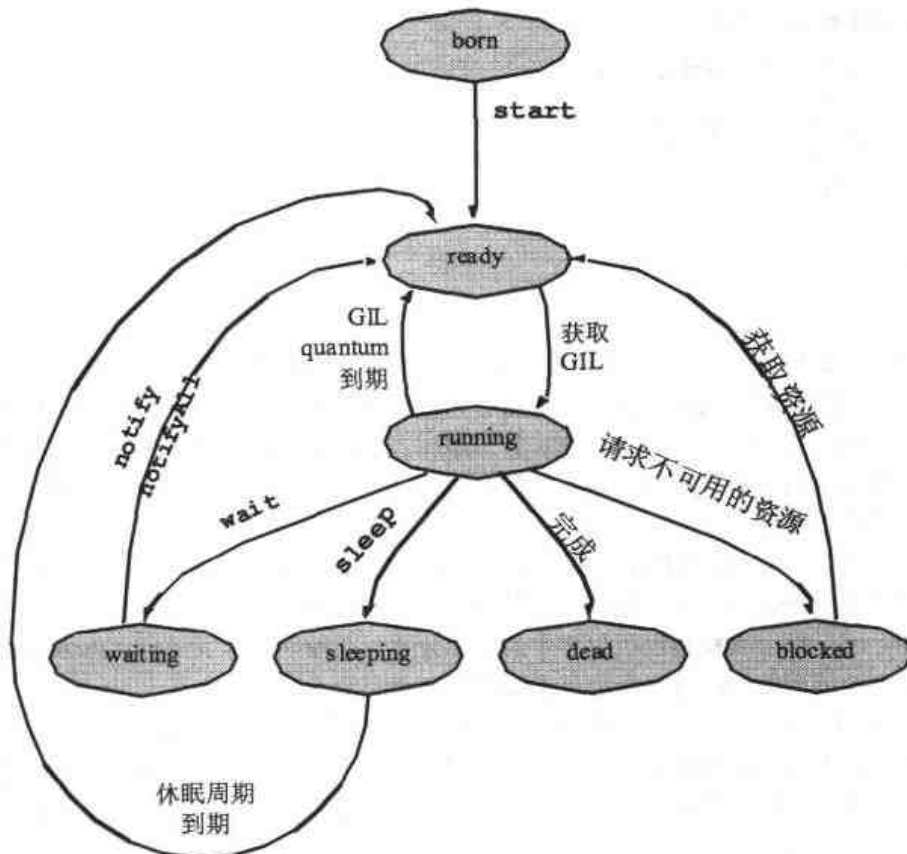


图 19.1 线程生命期状态图

多线程的 Python 程序使用 `threading` 模块的类和函数。程序可从 `threading.Thread` 类继承，并覆盖那个类的功能，从而定义线程。程序新建一个线程时，新线程将从“born”（诞生）状态开始它的生命期。线程将保持这个状态，直到程序调用线程的 `start` 方法，这会使线程进入“ready”（就绪）状态，有时也称为“runnable”（可运行）状态。另外，控制权会立即返回至调用线程（调用者）。之后，调用者可与已启动的线程以及程序中的其他任何线程并发执行。

常见编程错误 19.1 如果程序为已经退出 born 状态的线程调用 `start` 方法，方法就会引发 `AssertionError` 异常。

一个类从 `Thread` 继承时，派生类会覆盖基类的 `run` 方法。该方法实现了线程执行的任务。使用了线程的程序不调用那个线程的 `run` 方法。相反，当“ready”（就绪）线程首次获得 GIL 时，会执行它的 `run` 方法，成为一个“running”（正在运行）线程。`run` 方法会一直执行，直到线程引发一个未处理的异常，或者线程离开解释器。running 线程离开解释器时，线程会记住它的当前执行位置。以后重新进入解释器，线程会从该位置继续执行。线程惟一能获得 GIL 的状态就是“running”状态。由于任何原因而退出该状态，线程就会失去 GIL，使另一个线程可以执行。

`run` 方法返回或终止（比如遇到一个未进行捕捉的异常），就会进入“dead”（死亡）状态——解释器最终会对 dead 线程进行处置。如果 running 线程调用另一个线程的 `join`（加入）方法，running 线程会失去 GIL，并等待加入的方法死亡之后才会继续。注意，线程不可调用自己的 `join` 方法。`join` 的一个可

选参数是超时设置，这是一个浮点数，指定调用者要在多长时间（以秒为单位）内等待加入的线程结束。如果不为 `join` 传递参数，就表明调用者会无休止地等待目标线程死亡。这样的等待比较危险，可能导致两个特别严重的问题，即死锁和不确定延期。在死锁中，一个或多个线程遥遥无期地等待一个不可能发生的事件。在不确定延期后，一个或多个线程将延迟不可预测的时间。19.4 节将详细讨论死锁。

大多数程序都使用外部资源（比如网络连接和磁盘文件）来执行任务。如果线程请求的资源不可用，线程会进入“blocked”（暂停或阻塞）状态，直到资源再次可用。通常情况下，在线程发出一个 I/O 请求时，线程就可能进入 blocked 状态。线程发出 I/O 请求后（比如从磁盘上读入文件，或者将文件发送到打印机），会失去它的 GIL，并离开解释器。这样一来，在 I/O 执行期间，其他线程就可使用解释器。对于 I/O 密集型程序（工作主要由 I/O 任务构成），通常都能从多线程处理中获益，因为一个线程等候 I/O 的同时，程序中的其他线程可以使用解释器。这样便可更高效地利用处理器，而且有助于缩短程序的总体执行时间。I/O 操作结束后，在“blocked”状态等候它的线程会进入“ready”状态。之后，线程会试图重新获取 GIL。

“running”线程调用 `time.sleep` 函数后，会释放 GIL 并进入“sleeping”（休眠）状态。指定的休眠时间到期，“sleeping”线程会返回“ready”状态。即使解释器可用，“sleeping”线程也不能使用解释器。

程序中的线程通常共享数据。如果多个线程修改相同的数据，数据会被破坏，程序可能产生不可预料的结果。在这种程序中，可能需要同步对共享数据的访问。这意味着访问共享数据的每个线程首先必须获得与数据对应的一个同步对象锁。一旦线程处理完数据，就应释放同步对象，使其他线程能访问数据（同步的详情在 19.4 节~19.8 节讨论）。

有时因为一个程序的逻辑需求，正在运行的线程即使为共享数据获得了同步对象，也不能对其执行操作。在这种情况下，线程可调用同步对象的 `wait` 方法以主动释放对象。这会导致线程释放 GIL 并针对那个同步对象进入“waiting”状态。另一个线程调用同步对象的 `notify` 方法时，那个同步对象的一个“waiting”线程会变成“ready”状态。在重新获得 GIL 后，那个线程会恢复执行。“running”线程调用同步对象的 `notifyAll` 方法，处于“waiting”状态的每个线程都会变成“ready”状态。然后，解释器选择一个“ready”线程来执行。

`threading` 模块为程序提供了几种方式来获取与其线程有关的信息，其中包括它们的当前状态。`threading.currentThread` 函数会返回对当前正在运行的线程的一个引用。`threading.enumerate` 函数返回一个列表，其中包含 `Thread` 类的当前所有活动对象（即 `run` 方法开始但未终止的任何线程）。`threading.activeCount` 函数返回这个列表的长度。如果线程的 `start` 方法已被调用，而且线程没有死亡（即负责控制的 `run` 方法没有终止），线程方法 `isAlive` 会返回 1。`setName` 方法设置线程的名称，`getName` 则返回名称。为线程使用 `print` 语句会显示线程名称及其当前状态。本章将综合运用各种 `Thread` 方法和 `threading` 函数，获取和显示与线程有关的信息。

19.3 threading.Thread 示例

图 19.2 的程序定义一个从 `threading.Thread` 派生的类，并实例化那个类的对象，以演示如何创建线程。该程序还演示了 `time.sleep` 函数。程序中的 3 个线程都在休眠随机时间（1~5 秒）后，显示自己的名称。

`PrintThread` 类从 `threading.Thread` 继承，它包括 `sleepTime` 属性、一个构造函数以及一个 `run` 方法。`sleepTime` 属性（第 15 行）存储一个随机整数值，该值是在构造 `PrintThread` 对象时确定的。启动后，每个 `PrintThread` 对象都会休眠 `sleepTime` 所指定的时间，然后再输出自己的名称。

```
1 # Fig. 19.2: fig19_02.py
2 # Multiple threads printing at different intervals.
3
4 import threading
5 import random
6 import time
```

```

7
8 class PrintThread( threading.Thread ):
9     """Subclass of threading.Thread"""
10
11     def __init__( self, threadName ):
12         """Initialize thread, set sleep time, print data"""
13
14         threading.Thread.__init__( self, name = threadName )
15         self.sleepTime = random.randrange( 1, 6 )
16         print "Name: %s; sleep: %d" % \
17             ( self.getName(), self.sleepTime )
18
19     # overridden Thread run method
20     def run( self ):
21         """Sleep for 1-5 seconds"""
22
23         print "%s going to sleep for %s second(s)" % \
24             ( self.getName(), self.sleepTime )
25         time.sleep( self.sleepTime )
26         print self.getName(), "done sleeping"
27
28 thread1 = PrintThread( "thread1" )
29 thread2 = PrintThread( "thread2" )
30 thread3 = PrintThread( "thread3" )
31
32 print "\nStarting threads"
33
34 thread1.start()    # invokes run method of thread1
35 thread2.start()    # invokes run method of thread2
36 thread3.start()    # invokes run method of thread3
37
38 print "Threads started\n"

```

```

Name: thread1; sleep: 5
Name: thread2; sleep: 1
Name: thread3; sleep: 3

Starting threads
thread1 going to sleep for 5 second(s)
thread2 going to sleep for 1 second(s)
thread3 going to sleep for 3 second(s)
Threads started

thread2 done sleeping
thread3 done sleeping
thread1 done sleeping

```

图 19.2 按随机时间间隔打印线程

PrintThread 构造函数（第 11~17 行）首先调用基类构造函数，向其传递对象和线程名。线程名用 Thread 的关键字参数 name 指定。如果没有指定名称，程序就自动为线程指派一个不重复的名称，形如“Thread-*n*”，其中 *n* 是一个整数。然后，构造函数将 sleepTime 初始化成 1~5 之间的一个随机整数。接着，程序输出线程名和 sleepTime 的值，显示正在构造的特定 PrintThread 的值。

调用 PrintThread 的 start 方法（从 Thread 继承）时，PrintThread 对象会进入“ready”状态。解释器将切换到 PrintThread 对象，而线程进入“running”状态。如果这是线程首次进入“running”状态，会开始执行线程的 run 方法。run 方法（第 20~26 行）会显示一条消息，指出消息要进入休眠状态。然后调用 time.sleep 函数（第 25 行），使线程立即进入“sleeping”状态。经过 sleepTime 所指定的秒数后，线程会被唤醒，并再次进入“ready”状态，并等候进入解释器。PrintThread 对象重新进入“running”状态后，线程会从 sleep 调用之后（第 25 行）的下一个语句继续执行，以输出它的名称，并指出已经结束休眠。这样便结束了线程的 run 方法的执行，所以线程终止并进入“dead”状态。

程序的主要部分（第 28~38 行）创建了 3 个 PrintThread 对象，并为每个对象逐一调用 Thread 类的 start 方法，使所有 PrintThread 对象都转变成“ready”状态。之后，主程序的线程终止。然而，这个例子

会继续执行，直到最后一个 `PrintThread` 结束它的 `run` 方法。

19.4 线程同步

多个执行线程经常要处理共享数据。如果仅仅是读取共享数据，就不必禁止多个线程同时访问数据。然而，如果多个线程共享数据，而且有一个或多个线程修改数据，就可能出现无法预料的结果。两个线程同时更新数据，数据反映的是最近的更新。如果是数组或其他数据结构，几个线程同时更新它时，一部分数据会反映来自一个线程的信息，另一部分则反映来自另一个线程的信息。这样，程序就很难判断数据是否正确更新。通常将访问共享数据的代码区称为“临界区”。

为解决这个问题，可让访问共享数据的线程独占性地处理共享数据。在这段时间中，其他试图处理数据的线程应保持等待状态。具有独占访问权的线程结束数据处理后，应允许等候处理数据的一个线程介入。采取这种方式，访问共享数据的每个线程都禁止其他所有线程同时访问相同的数据。这称为“独占”或“线程同步”。

`threading` 模块提供了许多线程同步机制。最简单的同步机制是“锁”。锁对象用 `threading.RLock` 类创建，它定义了两个方法，即 `acquire`（获得）和 `release`（释放）。线程调用 `acquire` 方法，锁会进入“locked”（锁定）状态。每次只有一个线程可获得锁。如另一个线程试图对同一个锁对象调用 `acquire` 方法，操作系统会将那个线程转变成“blocked”状态，直到锁变得可用。拥有锁的线程调用 `release` 方法，锁会进入“unlocked”（解锁）状态。“blocked”的线程会收到一个通知，并可获得锁。如果多个锁处于“blocked”状态，所有线程都会先解除该状态。然后，操作系统选择一个线程来获得锁，再将其余线程变回“blocked”状态。

锁可限制对临界区的访问。在多线程程序中，线程必须先获得一个锁，再进入临界区；退出临界区时，则要释放锁。所以，如果一个线程正在临界区中执行，试图进入临界区的其他线程会被阻挡，直到原始线程退出临界区，而且释放了锁。

常见编程错误 19.2 必须将所有临界区都封闭在 `acquire` 和 `release` 调用之间。

但这只能提供最基本的同步级别。有时需要创建更复杂的线程，只在发生某些事件时才访问一个临界区（比如在某个数据值改变时）。这是通过“条件变量”来完成的。线程用条件变量监视一个对象的状态，或者发出事件通知。对象状态改变或事件发生时，处于 `blocked` 状态的线程会收到通知。本章后面要借助经典的“生产者/消费者”问题来讨论条件变量。在这个问题的解决方案中，牵涉到一个消费者线程，只有在收到一个生产者线程的通知时，它才会访问某个临界区，反之亦然。

条件变量用 `threading.Condition` 类创建。由于条件变量包含“基本锁”，所以它们提供了 `acquire` 和 `release` 方法。条件变量的其他方法还有 `wait` 和 `notify`。线程成功获得一个基本锁后，调用 `wait` 方法会导致调用线程释放这个锁，并进入“blocked”状态，直到另一个线程调用同一个条件变量的 `notify` 方法，从而将其唤醒。`notify` 方法可唤醒一个正在等待条件变量的线程；`notifyAll` 则唤醒所有正在等待的方法。

如果程序使用了锁、条件变量或其他同步机制，就有必要仔细检查，保证程序不会死锁。如果程序或线程永远处于“blocked”状态，就会发生死锁（即等待永远不会发生的一个事件或者永远用不了的一个资源）。例如，假定线程进入一个临界区，并打开某个文件。如果文件不存在，同时线程没有捕捉这个异常，线程就会在释放锁之前终止。其他试图访问该文件的所有线程都会死锁。它们在调用了锁的 `acquire` 方法后，就进入了遥遥无期的“blocked”状态。

常见编程错误 19.3 等待一个条件变量的线程必须用 `notify` 显式地唤醒，否则它会永远地等待下去，这可能会导致死锁。

测试和调试提示 19.1 保证每个 `wait` 调用都有一个对应的 `notify` 调用，以最终结束等待，也可调用 `notifyAll` 以策万全。

性能提示 19.2 通过同步来确保多线程程序的正确性，可能会减慢程序运行速度，这是由于锁造成了额外的开销，而且需要在线程的不同状态之间频繁切换。

19.5 生产者/消费者关系：无线程同步

在“生产者/消费者”(Producer/Consumer)关系中，应用程序的“生产者”部分生成数据，“消费者”部分则使用数据。在多线程生产者/消费者关系中，“生产者线程”调用一个“生产方法”来生成数据，并将数据放到名为“缓冲区”的共享内存区域。“消费者线程”则调用一个“消费方法”来读取数据。如果正在等待放入下一批数据的生产者线程发现消费者线程尚未从缓冲区读取上一批数据，生产者线程就会调用条件变量的 wait 方法；否则，消费者线程将无法看到上一批数据，那些数据会从该应用程序中丢失。消费者线程读取消息时，应调用条件变量的 notify 方法，使正在等待的生产者线程继续。如果消费者线程发现缓冲区为空，或上一批数据已经读取，就应调用条件变量的 wait 方法；否则，消费者线程会从缓冲区读入“垃圾”数据，或者重复处理以前的数据项——任何一种可能都会导致应用程序出现逻辑错误。生产者线程将下一批数据放入缓冲区时，生产者线程应调用条件变量的 notify 方法，让消费者线程继续。

在使用共享数据的多个线程之间，如果不进行访问同步，逻辑错误是怎样发生的呢？在这个假设的生产者/消费者关系中，生产者线程向一个“共享缓冲区”（由多个线程共享的内存位置）写入一系列数字(1~4)。消费者线程从共享缓冲区读取并显示这个数据。我们在程序输出中显示生产者所写入(生产)的值，以及消费者所读取(消费)的值。图 19.3~图 19.6 演示生产者和消费者访问一个共享内存单元(buffer)，同时不进行任何同步。消费者和生产者线程都要访问这个单元：生产者线程向单元写入；消费者线程从中读取。我们希望生产者线程写入共享单元的每个值都只能由消费者线程使用一次。然而，这个例子中的线程没有同步。因此，如果在消费者线程使用上一个数据之前，生产者线程将新数据放入共享单元，数据就会丢失。此外，在生产者线程产生下一个数据之前，如果消费者再次使用数据，会造成数据错误地重复。为演示这些可能性，下一个例子中的消费者线程将保存读取的所有值的总和。生产者线程产生从 1 到 4 的值。如果消费者线程只读取一次由生产者线程所产生的数据，总和应该是 10。但多次执行该程序，就会发现这个总和很少会是 10。另外，为了强调我们的观点，本例的生产者线程和消费者线程在每次执行任务之后，都要随机休眠一段时间按，最长 3 秒种。所以，我们不能准确地知道生产者线程何时写入一个新值，也不知道消费者线程何时读取一个值。图 19.3 演示了一个生产者(在图 19.4k 中定义)和一个消费者(在图 19.5 中定义)，它们访问一个共享内存单元(在图 19.6 中定义)，而且不进行任何同步。

```

1 # Fig. 19.3: fig19_03.py
2 # Multiple threads modifying shared object.
3
4 from UnsynchronizedInteger import UnsynchronizedInteger
5 from ProduceInteger import ProduceInteger
6 from ConsumeInteger import ConsumeInteger
7
8 # initialize integer and threads
9 number = UnsynchronizedInteger()
10 producer = ProduceInteger("Producer", number, 1, 4)
11 consumer = ConsumeInteger("Consumer", number, 4)
12
13 print "Starting threads...\n"
14
15 # start threads
16 producer.start()
17 consumer.start()
18
19 # wait for threads to terminate
20 producer.join()
21 consumer.join()
22

```

```
23 print "\nAll threads have terminated."
```

```
Starting threads...
Consumer reads -1
Consumer reads -1
Producer writes 1
Consumer reads 1
Producer writes 2
Consumer reads 2
Consumer read values totaling: 1.
Terminating Consumer.
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.

All threads have terminated.
```

```
Starting threads...
Producer writes 1
Producer writes 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer reads 4
Consumer reads 4
Consumer read values totaling: 15.
Terminating Consumer.

All threads have terminated.
```

```
Starting threads...
Producer writes 1
Consumer reads 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer read values totaling: 10.
Terminating Consumer.

All threads have terminated.
```

图 19.3 用于修改未同步共享对象的线程

```
1 # Fig. 19.4: ProduceInteger.py
2 # Integer-producing class.
3
4 import threading
5 import random
6 import time
7
8 class ProduceInteger( threading.Thread ):
9     """Thread to produce integers"""
10
11     def __init__( self, threadName, sharedObject, begin, end ):
12         """Initialize thread, set shared object"""
13
14         threading.Thread.__init__( self, name = threadName )
15         self.sharedObject = sharedObject
```

```

16     self.begin = begin
17     self.end = end
18
19     def run( self ):
20         """Produce integers in given range at random intervals"""
21
22         for i in range( self.begin, ( self.end + 1 ) ):
23             time.sleep( random.randrange( 4 ) )
24             self.sharedObject.set( i )
25
26         print "%s done producing." % self.getName()
27         print "Terminating %s." % self.getName()

```

图 19.4 整数生产者线程

```

1 # Fig. 19.5: ConsumeInteger.py
2 # Integer-consuming queue.
3
4 import threading
5 import random
6 import time
7
8 class ConsumeInteger( threading.Thread ):
9     """Thread to consume integers"""
10
11     def __init__( self, threadName, sharedObject, amount ):
12         """Initialize thread, set shared object"""
13
14         threading.Thread.__init__( self, name = threadName )
15         self.sharedObject = sharedObject
16         self.amount = amount
17
18     def run( self ):
19         """Consume given amount of values at random time intervals"""
20
21         sum = 0 # total sum of consumed values
22
23         # consume given amount of values
24         for i in range( self.amount ):
25             time.sleep( random.randrange( 4 ) )
26             sum += self.sharedObject.get()
27
28         print "%s read values totaling: %d." % \
29             ( self.getName(), sum )
30         print "Terminating %s." % self.getName()

```

图 19.5 整数消费者线程

```

1 # Fig. 19.6: UnsynchronizedInteger.py
2 # Unsynchronized access to an integer.
3
4 import threading
5
6 class UnsynchronizedInteger:
7     """Class that provides unsynchronized access to an integer"""
8
9     def __init__( self ):
10         """Initialize integer to -1"""
11
12         self.buffer = -1
13
14     def set( self, newNumber ):
15         """Set value of integer"""
16
17         print "%s writes %d" % \
18             ( threading.currentThread().getName(), newNumber )
19         self.buffer = newNumber
20
21     def get( self ):
22         """Get value of integer"""

```



```

23
24     tempNumber = self.buffer
25     print "%s reads %d" % \
26         ( threading.currentThread().getName(), tempNumber )
27
28     return tempNumber

```

图 19.6 未同步的整数值类

图 19.3 创建名为 `number` 的共享 `UnsynchronizedInteger` 对象(第 9 行),并把它作为参数传给 `producer` 对象的构造函数(第 10 行),`producer` 是一个 `ProduceInteger` 对象;再传给 `consumer` 对象的构造函数(第 11 行),`consumer` 是一个 `ConsumeInteger` 对象。接着,程序调用 `product` 和 `consumer` 对象的线程方法 `start`,把它们转变成“ready”状态(第 16~17 行)。第 20~21 行调用线程方法 `join`,保证主程序在等候两个线程都终止之后才会继续。注意两个线程终止后,才会执行第 23 行。

`ProduceInteger` 类(图 19.4)是 `threading.Thread` 的一个子类,它包括属性(`sharedObject`, `begin` 和 `end`)、一个构造函数(第 11~17 行)以及一个 `run` 方法(第 19~27 行)。构造函数负责初始化 `sharedObject` 属性(第 15 行),令其引用作为参数传递的 `UnsynchronizedInteger` 对象。

`ProduceInteger` 的 `run` 方法包括一个 `for` 结构,它从 `begin` 循环到 `end`(第 22~24 行)。每次重复时,都首先调用 `time.sleep` 函数,以便将 `ProduceInteger` 对象转变成“sleeping”(休眠)状态,并随机等待一段时间,约为 0~3 秒。线程唤醒后,会调用共享对象的 `set` 方法(第 24 行),并传递控制变量 `i` 的值,目的是设置共享对象的数据成员。注意共享对象的 `set` 方法是从 `ProduceInteger` 线程调用的。换言之,从中调用方法的线程执行在方法定义中指定的语句。循环结束时,`ProduceInteger` 线程会在命令窗口显示一行文本,指出已结束了数据生成,并准备终止。之后,线程实际终止(即线程死亡)。

`ConsumeInteger` 类(图 19.5)是 `threading.Thread` 类的一个子类,其中包括属性(`sharedObject` 和 `amount`)、一个构造函数(第 11~16 行)和一个 `run` 方法(第 18~30 行)。构造函数初始化 `sharedObject` 属性,令其引用作为参数传递的 `UnsynchronizedInteger` 对象(第 15 行)。

`ConsumeInteger` 类的 `run` 方法包括一个 `for` 结构,它循环 `amount` 所指定的次数,以便从 `sharedObject` 引用的 `UnsynchronizedInteger` 对象读取值(第 24~26 行)。每次循环,都要调用 `time.sleep` 函数,使 `ConsumeInteger` 对象随机休眠一段对间,约为 0~3 秒。接着,线程调用 `get` 方法,获得共享对象的数据成员的值。同样要注意共享对象的 `set` 方法是从 `ConsumeInteger` 线程调用的。从中调用方法的线程执行方法定义中指定的语句。之后,线程将 `get` 返回的值加到变量 `sum` 上(第 26 行)。循环结束后,`ConsumeInteger` 线程在命令窗口显示一行文本,指出它已结束使用数据,并准备终止。然后,线程实际地终止(即线程死亡)。

在图 19.6 中,`UnsynchronizedInteger` 类包括 `buffer` 属性(第 12 行)、`set` 方法(14~19 行)以及 `get` 方法(第 21~28 行)。`set` 和 `get` 方法不同步对 `buffer` 属性的访问。理想情况下,我们希望 `ProduceInteger` 对象产生的每个值分别供 `ConsumeInteger` 对象使用一次。但是,图 19.3 的示范输出证明,有的值已经丢失了(消费者根本没有看到),有的值则由消费者多次取得。一切都是因为没有进行同步。

事实上,`get` 方法必须执行一些额外工作,才能保证在输出中准确反映数据成员的值。第 24 行将数据成员 `buffer` 的值指派给变量 `tempNumber`。然后,第 25~28 行利用这个值打印消息并返回值。如果不像这样使用一个临时变量,就可能出现以下情况:消费者调用 `get` 方法,并打印一条消息来显示数据成员的值。随后,解释器可能先将消费者线程移出,再将生产者线程移入。然后,生产者线程可能调用 `set` 方法来更改 `buffer` 值。最终,解释器移入消费者,`get` 方法返回一个值,该值与消费者移出之前所打印的值不同。

理想情况下,`Producer` 对象产生的每个值都只由 `Consumer` 对象使用一次。但分析图 19.3 的输出,会发现在生产者尚未向共享缓冲区中放入任何值之前,消费者就已经获得了两次值,而且每次都是 -1。此外,值 3 和值 4 根本没有使用过,因为在生产者有机会产生那些值之前,消费者就已结束执行,所以那些值会被丢失。在第二个输出中,丢失的值是 1 和 2,因为在消费者线程读取值 1 之前,就已经生成了值 1、2 和 3。此外,值 4 被使用了 3 次。最后一个示范输出证明在运气好的时候,也许能获得正确输

出，即生产者产生的每个值都只由消费者使用一次。这个例子清楚地证明了多个线程访问共享数据时，必须小心加以控制，否则程序会产生不正确的结果。

为解决丢失数据和多次使用相同数据的问题，图 19.7~图 19.8 将针对并发执行的生产者和消费者线程，利用条件变量和前文提过的 `acquire`、`release`、`wait` 和 `notify` 方法，使这些线程在访问用于处理共享数据的代码时同步。如果操作共享对象的线程用同一个条件来执行同步，每个线程就必须先获得条件变量的锁，然后才能实际地操作共享对象。在特定条件变量上，每次只有一个线程获得锁。这使得其他线程不能同时获得用于那个条件变量的锁。

19.6 生产者/消费者关系：有线程同步

下一个程序（图 19.7 和图 19.8）演示生产者和消费者访问共享内存单元，同时进行线程同步，让每个值只由消费者使用一次。示例程序还保证消费者首先等待生产者执行完毕。图 19.7 和图 19.3 的区别在于，它向生产者和消费者传递 `SynchronizedInteger` 类的一个对象，而且第 15~17 行打印表头和 `SynchronizedInteger` 的初始状态。`ProduceInteger` 和 `ConsumeInteger` 类与前一小节所用的完全相同，这里不再展示。

`SynchronizedInteger` 类（图 19.8）包含 3 个属性，分别是 `buffer`、`occupiedBufferCount` 和 `threadCondition`。其中，`occupiedBufferCount` 属性帮助 `SynchronizedInteger` 对象判断应该由生产者还是消费者来使用 `buffer`。`threadCondition` 属性是 `Condition` 类的一个对象，负责维护锁，防止多个线程同时使用 `buffer`。`Condition` 对象（也称为“条件变量”）还包含一些方法，以便供线程不能执行任务或者结束任务时使用，后文即将讨论这些方法。

`set` 方法（第 16~39 行）使用 `occupiedBufferCount` 和 `threadCondition` 判断调用方法的线程是否能向共享内存位置写入。`get` 方法（第 41~66 行）使用 `occupiedBufferCount` 和 `threadCondition` 判断调用者线程是否能从共享内存位置读取。`displayState` 方法（第 68~72 行）显示一条消息（当前操作），同时显示 `buffer` 和 `occupiedBufferCount` 的当前值。

```

1 # Fig. 19.7: fig19_07.py
2 # Multiple threads modifying shared object.
3
4 from SynchronizedInteger import SynchronizedInteger
5 from ProduceInteger import ProduceInteger
6 from ConsumeInteger import ConsumeInteger
7
8 # initialize number and threads
9 number = SynchronizedInteger()
10 producer = ProduceInteger( "Producer", number, 1, 4 )
11 consumer = ConsumeInteger( "Consumer", number, 4 )
12
13 print "Starting threads...\n"
14
15 print "%-35s %-9s%2s\n" % \
16     ( "Operation" , "Buffer", "Occupied Count" )
17 number.displayState( "Initial state" )
18
19 # start threads
20 producer.start()
21 consumer.start()
22
23 # wait for threads to terminate
24 producer.join()
25 consumer.join()
26
27 print "\nAll threads have terminated."

```

Starting threads...

Operation	Buffer	Occupied Count
Initial state	-1	0
Producer writes 1	1	1
Consumer reads 1	1	0
Consumer tries to read. Buffer empty. Consumer waits.	1	0
Producer writes 2	2	1
Consumer reads 2	2	0
Producer writes 3	3	1
Producer tries to write. Buffer full. Producer waits.	3	1
Consumer reads 3	3	0
Producer writes 4	4	1
Producer done producing. Terminating Producer.		
Consumer reads 4	4	0

Consumer read values totaling: 10.
Terminating Consumer.

All threads have terminated.

Starting threads...

Operation	Buffer	Occupied Count
Initial state	-1	0
Consumer tries to read. Buffer empty. Consumer waits.	-1	0
Producer writes 1	1	1
Consumer reads 1	1	0
Producer writes 2	2	1
Consumer reads 2	2	0
Consumer tries to read. Buffer empty. Consumer waits.	2	0
Producer writes 3	3	1
Consumer reads 3	3	0
Producer writes 4	4	1
Producer done producing. Terminating Producer.		
Consumer reads 4	4	0

Consumer read values totaling: 10.
Terminating Consumer.

All threads have terminated.

```

Starting threads...

```

Operation	Buffer	Occupied Count
Initial state	-1	0
Producer writes 1	1	1
Consumer reads 1	1	0
Producer writes 2	2	1
Consumer reads 2	2	0
Producer writes 3	3	1
Consumer reads 3	3	0
Producer writes 4	4	1
Producer done producing. Terminating Producer.		
Consumer reads 4	4	0
Consumer read values totaling: 10. Terminating Consumer.		
All threads have terminated.		

图 19.7 修改同步共享对象的线程

```

1 # Fig. 19.8: SynchronizedInteger.py
2 # Synchronized access to an integer with condition variable.
3
4 import threading
5
6 class SynchronizedInteger:
7     """Class that provides synchronized access to an integer"""
8
9     def __init__( self ):
10         """Initialize integer, buffer count and condition variable"""
11
12         self.buffer = -1
13         self.occupiedBufferCount = 0 # number of occupied buffers
14         self.threadCondition = threading.Condition()
15
16     def set( self, newNumber ):
17         """Set value of integer--blocks until lock acquired"""
18
19         # block until lock released then acquire lock
20         self.threadCondition.acquire()
21
22         # while not producer's turn, release lock and block
23         while self.occupiedBufferCount == 1:
24             print "%s tries to write." % \
25                 threading.currentThread().getName()
26             self.displayState( "Buffer full. " + \
27                 threading.currentThread().getName() + " waits." )
28             self.threadCondition.wait()
29
30         # (lock has now been re-acquired)
31
32         self.buffer = newNumber # set new buffer value
33         self.occupiedBufferCount += 1 # allow consumer to consume
34
35         self.displayState( "%s writes %d" % \
36             ( threading.currentThread().getName(), newNumber ) )
37
38         self.threadCondition.notify() # wake up a waiting thread
39         self.threadCondition.release() # allow lock to be acquired

```

```

40
41 def get( self ):
42     """Get value of integer--blocks until lock acquired"""
43
44     # block until lock released then acquire lock
45     self.threadCondition.acquire()
46
47     # while producer's turn, release lock and block
48     while self.occupiedBufferCount == 0:
49         print "%s tries to read." % \
50             threading.currentThread().getName()
51         self.displayState( "Buffer empty. " + \
52             threading.currentThread().getName() + " waits." )
53         self.threadCondition.wait()
54
55     # (lock has now been re-acquired)
56
57     tempNumber = self.buffer
58     self.occupiedBufferCount -= 1 # allow producer to produce
59
60     self.displayState( "%s reads %d" % \
61         ( threading.currentThread().getName(), tempNumber ) )
62
63     self.threadCondition.notify() # wake up a waiting thread
64     self.threadCondition.release() # allow lock to be acquired
65
66     return tempNumber
67
68 def displayState( self, operation ):
69     """Display current state"""
70
71     print "%-35s %-9s%-2s\n" % \
72         ( operation, self.buffer, self.occupiedBufferCount )

```

图 19.8 同步的整数值类

第 12~13 行分别将 `buffer` 和 `occupiedBufferCount` 属性初始化为 -1 和 0，目的是指出缓冲区当前为空。第 14 行调用 `threading.Condition` 构造函数，从而创建线程条件变量。这会为条件变量创建一个新锁。`threadCondition` 变量保护对 `occupiedBufferCount` 和 `buffer` 属性的访问。

如果 `occupiedBufferCount` 为 0，表明没有新产生的值，所以生产者能将一个值放入变量 `buffer`。然而，这也表明消费者当前不能读取 `buffer` 的值。如果 `occupiedBufferCount` 为 1，表明 `buffer` 包含一个新产生的值。该值必须先由消费者读取，否则生产者不能放入另一个新值。

`ProduceInteger` 线程调用 `set` 方法（第 16~39 行）时，线程获得条件变量的一个锁（第 20 行）。`while` 结构（第 23~28 行）检测 `occupiedBufferCount`，判断它是否为 1（`buffer` 为满）。如果是，生产者就显示当前状态，并调用条件变量的 `wait` 方法（第 28 行）。这会把调用 `set` 方法的 `ProduceInteger` 线程对象转变成 `threadCondition` 维护的“waiting”状态并释放锁。之后，其他线程可以访问 `SynchronizedInteger` 对象。

`ProduceInteger` 线程将保持“waiting”状态，直至收到可以继续的通知，之后就进入“ready”状态。`ProduceInteger` 线程重新进入“running”状态时，对象会显式地重新获取 `threadCondition` 的锁，`set` 方法继续执行 `while` 结构中 `wait` 之后的下一个语句。没有更多的语句，所以程序重新求值 `while` 条件。如果条件为 `false`（即 `occupiedBufferCount` 等于 0），`while` 结构终止。然后，第 32 行设置 `buffer` 的当前值。第 33 行使 `occupiedBufferCount` 自增，表明共享内存现在是满的（也就是说，消费者能读取值，生产者不能指派另一个值）。第 35~36 行调用 `displayState` 方法以显示新产生的值。第 38 行调用 `threadCondition` 的 `notify` 方法。如果 `threadCondition` 维护着任何处于 `waiting` 状态的等待线程，那么处于“waiting”状态的一个线程会进入“ready”状态。它现在可再次尝试自己的任务（进入解释器之后）。然后，第 39 行调用 `threadCondition` 的 `release` 方法以释放锁（使新的 `ready` 线程能够获得它），`set` 方法则返回至它的调用者。

常见编程错误 19.4 `Condition` 的 `notify` 方法不会释放基本锁。忘记调用 `Condition` 对象的 `release` 方法可能导致死锁。

get 方法和 set 方法以类似的方式实现。ConsumeInteger 线程调用 get 方法时, 该方法获得 threadCondition 的一个锁。while 结构 (第 48~53 行) 判断 occupiedBufferCount 变量是否为 0 (即没什么数据可以使用)。如果是, 线程就显示当前状态, 并调用 threadCondition 的 wait 方法。这使调用 get 方法的 ConsumeInteger 线程进入由 threadCondition 维护的“waiting”状态, 并释放锁, 使其他线程能访问它。ConsumeInteger 线程保持“waiting”状态, 直到收到可以继续的通知。之后, 它会进入“ready”状态, 并等待进入解释器。ConsumeInteger 线程重新进入“running”状态后, set 方法重新获得 threadCondition 的锁, 而 set 方法在 while 结构中继续执行 wait 之后的下一个语句。由于没有更多的语句, 所以程序再次检查 while 条件。如果 occupiedBufferCount 为 1, 就将 buffer 的值存储到变量 tempNumber 中 (第 57 行)。注意, buffer 值只取得一次, 并存储到变量 tempNumber 中 (同时仍在临界区中)。线程退出临界区后, buffer 可能被另一个线程修改。第 66 行新添加的代码, 在 while 循环中, 当 occupiedBufferCount 为 0 时, 调用 threadCondition 的 wait 方法, 使线程进入“waiting”状态, 并释放锁。当有其他线程调用 set 方法时, 线程会收到通知, 并重新进入“ready”状态, 等待进入解释器。

有足够单元来处理任何“额外”产生的数据项。在图 19.9 的示范输出中，注意只有在生产者插入一个值之后，消费者才能成功获取它。

`ProduceToQueue` 类（图 19.10）类似于图 19.4 的 `ProduceInteger` 类，不过稍有改动，以便访问由 `ProduceToQueue` 和 `ConsumeFromQueue`（图 19.11）线程共享的一个队列。`ProduceToQueue` 类的 `run` 方法由一个 `for` 结构（第 20~23 行）构成，它循环 10 次，将范围在 11~20 的值放入队列。每次循环时，都要首先调用 `time.sleep` 函数，使 `ProduceToQueue` 线程随机休眠 0~3 秒。线程唤醒后，会调用 `sharedObject` 队列的 `put` 方法（第 23 行），并向其传递控制变量 `i` 的值，以便将值插入队列。循环结束后，`ProduceToQueue` 线程在命令窗口显示一行文本，指出它已结束数据产生，并准备终止线程。然后，线程实际终止。

`ConsumeFromQueue` 类（图 19.11）类似于 `ConsumeInteger` 类（图 19.5）。`ConsumeFromQueue` 类的 `run` 方法由一个 `for` 结构（第 24~30 行）构成，它循环 10 次，从 `sharedObject` 所引用的 `Queue` 对象读取值。每次循环，都要调用 `time.sleep` 函数，使 `ConsumeFromQueue` 线程随机休眠 0~3 秒。然后，线程调用 `sharedObject` 的 `get` 方法，获得队列中的下一个值。如队列为空，线程会暂停，直到有可用的值。获得值后，线程打印消息以显示取得的值（第 29 行）。然后，线程将 `current` 值（`get` 所返回的值）加到变量 `sum` 上（第 30 行）。循环终止后，`ConsumeFromQueue` 线程在命令窗口显示一行文本，指出它已结束数据使用，并准备终止线程。然后，线程实际终止。

在图 19.9 的示范输出中，注意队列中无值可用时，`ConsumeFromQueue` 线程会自动等待 `ProduceToQueue` 线程产生一个新值。之后，`ConsumeFromQueue` 线程才会实际地完成它的读取操作。

```

1 # Fig. 19.9: fig19_09.py
2 # Multiple threads producing/consuming values.
3
4 from Queue import Queue
5 from ProduceToQueue import ProduceToQueue
6 from ConsumeFromQueue import ConsumeFromQueue
7
8 # initialize number and threads
9 queue = Queue()
10 producer = ProduceToQueue( "Producer", queue )
11 consumer = ConsumeFromQueue( "Consumer", queue )
12
13 print "Starting threads...\n"
14
15 # start threads
16 producer.start()
17 consumer.start()
18
19 # wait for threads to terminate
20 producer.join()
21 consumer.join()
22
23 print "\nAll threads have terminated."

```

```

Starting threads...

Producer adding 11 to queue
Producer adding 12 to queue
Consumer attempting to read 11...
Consumer read 11
Consumer attempting to read 12...
Consumer read 12
Producer adding 13 to queue
Consumer attempting to read 13...
Consumer read 13
Producer adding 14 to queue
Consumer attempting to read 14...
Consumer read 14
Consumer attempting to read 15...
Producer adding 15 to queue
Consumer read 15
Consumer attempting to read 16...
Producer adding 16 to queue
Consumer read 16

```

```

Consumer attempting to read 17...
Producer adding 17 to queue
Consumer read 17
Producer adding 18 to queue
Producer adding 19 to queue
Consumer attempting to read 18...
Consumer read 18
Consumer attempting to read 19...
Consumer read 19
Producer adding 20 to queue
Producer finished producing values
Terminating Producer
Consumer attempting to read 20...
Consumer read 20
Consumer retrieved values totaling: 155
Terminating Consumer

All threads have terminated.

```

图 19.9 线程通过队列产生/使用值

```

1 # Fig. 19.10: ProduceToQueue.py
2 # Integer-producing class.
3
4 import threading
5 import random
6 import time
7
8 class ProduceToQueue( threading.Thread ):
9     """Thread to produce integers"""
10
11     def __init__( self, threadName, queue ):
12         """Initialize thread, set shared queue"""
13
14         threading.Thread.__init__( self, name = threadName )
15         self.sharedObject = queue
16
17     def run( self ):
18         """Produce integers in range 11-20 at random intervals"""
19
20         for i in range( 11, 21 ):
21             time.sleep( random.randrange( 4 ) )
22             print "%s adding %s to queue" % ( self.getName(), i )
23             self.sharedObject.put( i )
24
25         print self.getName(), "finished producing values"
26         print "Terminating", self.getName()

```

图 19.10 整数生产者线程

```

1 # Fig. 19.11: ConsumeFromQueue.py
2 # Integer-consuming queue.
3
4 import threading
5 import random
6 import time
7
8 class ConsumeFromQueue( threading.Thread ):
9     """Thread to consume integers"""
10
11     def __init__( self, threadName, queue ):
12         """Initialize thread, set shared queue"""
13
14         threading.Thread.__init__( self, name = threadName )
15         self.sharedObject = queue
16
17     def run( self ):
18         """Consume 10 values at random time intervals"""
19
20         sum = 0          # total sum of consumed values

```



```

21     current = 10          # last value retrieved
22
23     # consume 10 values
24     for i in range( 10 ):
25         time.sleep( random.randrange( 4 ) )
26         print "%s attempting to read %s..." % \
27             ( self.getName(), current + 1 )
28         current = self.sharedObject.get()
29         print "%s read %s" % ( self.getName(), current )
30         sum += current
31
32     print "%s retrieved values totaling: %d" % \
33         ( self.getName(), sum )
34     print "Terminating", self.getName()

```

图 19.11 整数消费者线程

19.8 生产者/消费者关系：循环缓冲区

图 19.9~图 19.11 的程序允许生产者产生值的速度超过消费者使用值的速度。图 19.12 和图 19.13 演示了另一种数据结构，它可尽量缩短共享相同资源的不同线程的等待时间，并以同样的相对速度运行。

“循环缓冲区”（Circular Buffer）提供了额外的缓冲区，生产者可在其中放入值，消费者可从中获取那些值。假定缓冲区作为一个数组来实现。生产者和消费者从数组头开始工作。任何线程一旦抵达数组尾，就回到数组的第一个元素，执行下一个任务。如果生产者生成值的速度暂时快于消费者使用值的速度，生产者可将附加的值写入额外的缓冲区（如单元可用）。这样一来，即使消费者尚未准备好接收当前产生的值，生产者也能执行它的任务，无需等待。类似地，如果消费者任务使用值的速度快于生产者生成新值的速度，消费者可从缓冲区读取附加的值（如果有的话）。这样一来，即使生产者尚未准备好产生附加的值，消费者仍可执行它的任务，而无需等待。

注意假如生产者和消费者以不同速度运行，也许不适合使用循环缓冲区。如消费者总是比生产者快，只包含一个位置（单元）的缓冲区就已足够，更多的内存位置纯属浪费。如果生产者总要快一些，缓冲区内就需要包含数量不限的位置，以吸收不断生成的额外数据。

使用循环缓冲区时，关键在于定义足够的额外单元，以处理预料之中的、“额外”生成的数据。如果在一段时间之后，我们发现生产者产生的数据总比消费者使用的数据多出 3 倍，就可定义至少包含 3 个单元的缓冲区，以便处理额外产生的数据。缓冲区不能太小，否则线程会等待更长的时间。但也不能太大以至于浪费内存。

性能提示 19.3 即使使用循环缓冲区，生产者线程也可能填满缓冲区，迫使生产者线程等待消费者线程去使用一个数值，以腾出缓冲区中的一个元素。类似地，如果缓冲区总是空的，消费者线程就必须等待生产者产生另一个值。使用循环缓冲区时，关键在于优化缓冲区大小，尽可能缩短线程等待时间。

图 19.12 和图 19.13 演示一个生产者和消费者访问同步循环缓冲区（本例是一个含有 3 个单元的共享列表）的情况。如列表包含一个或多个值，消费者就只使用其中的一个；如列表包含一个或多个可用单元，生产者就只产生一个值。图 19.12 使用 `ProduceInteger` 类（图 19.4）和 `ConsumeInteger` 类（图 19.5）稍微修改过的版本来产生和使用范围在 11~20 之间（而不是 1~4 之间）的值。注意在示范输出中，当第 3 个值放到缓冲区的第 3 个元素后，第 4 个值将在列表起始处插入，这样便获得了循环缓冲区的效果。

```

1  # Fig. 19.12: fig19_12.py
2  # Show multiple threads modifying shared object.
3
4  from CircularBuffer import CircularBuffer
5  from ProduceInteger import ProduceInteger
6  from ConsumeInteger import ConsumeInteger
7
8  # initialize number and threads
9  buffer = CircularBuffer()
10 producer = ProduceInteger( "Producer", buffer, 11, 20 )

```

```

11 consumer = ConsumeInteger( "Consumer", buffer, 10 )
12
13 print "Starting threads...\n"
14
15 buffer.displayState()
16
17 # start threads
18 producer.start()
19 consumer.start()
20
21 # wait for threads to terminate
22 producer.join()
23 consumer.join()
24
25 print "\nAll threads have terminated."

```

Starting threads...

(buffers occupied: 0)

buffers: -1 -1 -1

WR

Producer writes 11 (buffers occupied: 1)

buffers: 11 -1 -1

R W

Consumer reads 11 (buffers occupied: 0)

buffers: 11 -1 -1

WR

All buffers empty. Consumer waits.

Producer writes 12 (buffers occupied: 1)

buffers: 11 12 -1

R W

Consumer reads 12 (buffers occupied: 0)

buffers: 11 12 -1

WR

All buffers empty. Consumer waits.

Producer writes 13 (buffers occupied: 1)

buffers: 11 12 13

W R

Consumer reads 13 (buffers occupied: 0)

buffers: 11 12 13

WR

All buffers empty. Consumer waits.

Producer writes 14 (buffers occupied: 1)

buffers: 14 12 13

R W

Consumer reads 14 (buffers occupied: 0)

buffers: 14 12 13

WR

Producer writes 15 (buffers occupied: 1)

buffers: 14 15 13

R W

```

Producer writes 16 (buffers occupied: 2)
buffers: 14 15 16
-----
      W      R

Consumer reads 15 (buffers occupied: 1)
buffers: 14 15 16
-----
      W      R

Producer writes 17 (buffers occupied: 2)
buffers: 17 15 16
-----
      W      R

Producer writes 18 (buffers occupied: 3)
buffers: 17 18 16
-----
              MR

All buffers full. Producer waits.
Consumer reads 16 (buffers occupied: 2)
buffers: 17 18 16
-----
      R      W

Producer writes 19 (buffers occupied: 3)
buffers: 17 18 19
-----
              MR

All buffers full. Producer waits.
Consumer reads 17 (buffers occupied: 2)
buffers: 17 18 19
-----
      W      R

Producer writes 20 (buffers occupied: 3)
buffers: 20 18 19
-----
              MR

Producer done producing.
Terminating Producer.

Consumer reads 18 (buffers occupied: 2)
buffers: 20 18 19
-----
      W      R

Consumer reads 19 (buffers occupied: 1)
buffers: 20 18 19
-----
      R      W

Consumer reads 20 (buffers occupied: 0)
buffers: 20 18 19
-----
              MR

Consumer read values totaling: 165.
Terminating Consumer.

All threads have terminated.

```

图 19.12 线程修改同步循环缓冲区

1 # Fig. 19.13: CircularBuffer.py

```

2 # Synchronized circular buffer of integer values
3
4 import threading
5
6 class CircularBuffer:
7
8     def __init__( self ):
9         """Set buffer, count, locations and condition variable"""
10
11         # each element in list is a buffer
12         self.buffer = [ -1, -1, -1 ]
13
14         self.occupiedBufferCount = 0 # count of occupied buffers
15         self.readLocation = 0        # current reading index
16         self.writeLocation = 0       # current writing index
17
18         self.threadCondition = threading.Condition()
19
20     def set( self, newNumber ):
21         """Set next buffer index value--blocks until lock acquired"""
22
23         # block until lock released then acquire lock
24         self.threadCondition.acquire()
25
26         # while all buffers are full, release lock and block
27         while self.occupiedBufferCount == len( self.buffer ):
28             print "All buffers full. %s waits." % \
29                 threading.currentThread().getName()
30             self.threadCondition.wait()
31
32         # (there is an empty buffer, lock has been re-acquired)
33
34         # place value in writeLocation of buffer
35         # print string indicating produced value
36         self.buffer[ self.writeLocation ] = newNumber
37         print "%s writes %d " % \
38             ( threading.currentThread().getName(), newNumber ),
39
40         # produced value, so increment number of occupied buffers
41         self.occupiedBufferCount += 1
42
43         # update writeLocation for future write operation
44         # add current state to output
45         self.writeLocation = ( self.writeLocation + 1 ) % \
46             len( self.buffer )
47         self.displayState()
48
49         self.threadCondition.notify() # wake up a waiting thread
50         self.threadCondition.release() # allow lock to be acquired
51
52     def get( self ):
53         """Get next buffer index value--blocks until lock acquired"""
54
55         # block until lock released then acquire lock
56         self.threadCondition.acquire()
57
58         # while all buffers are empty, release lock and block
59         while self.occupiedBufferCount == 0:
60             print "All buffers empty. %s waits." % \
61                 threading.currentThread().getName()
62             self.threadCondition.wait()
63
64         # (there is a full buffer, lock has been re-acquired)
65
66         # obtain value at current readLocation
67         # print string indicating consumed value
68         tempNumber = self.buffer[ self.readLocation ]
69         print "%s reads %d " % ( threading.currentThread().getName(),
70                                 tempNumber ),
71
72         # consumed value, so decrement number of occupied buffers

```

```

73     self.occupiedBufferCount -= 1
74
75     # update readLocation for future read operation
76     # add current state to output
77     self.readLocation = ( self.readLocation + 1 ) % \
78         len( self.buffer )
79     self.displayState()
80
81     self.threadCondition.notify()    # wake up a waiting thread
82     self.threadCondition.release()   # allow lock to be acquired
83
84     return tempNumber
85
86 def displayState( self ):
87     """Display current state"""
88
89     # display first line of state information
90     print "(buffers occupied: %d)" % self.occupiedBufferCount
91     print "buffers: ",
92
93     for item in self.buffer:
94         print " %d " % item,
95
96     # display second line of state information
97     print "\n      ",
98
99     for item in self.buffer:
100         print "---- ",
101
102     # display third line of state information
103     print "\n      ",
104
105     for i in range( len( self.buffer ) ):
106
107         if ( i == self.writeLocation ) and \
108             ( self.writeLocation == self.readLocation ):
109             print " WR ",
110         elif ( i == self.writeLocation ):
111             print " W  ",
112         elif ( i == self.readLocation ):
113             print " R  ",
114         else:
115             print "    ",
116
117     print "\n"

```

图 19.13 包含整数的同步循环缓冲区

CircularBuffer 类（图 19.13）包含 5 个属性。其中，buffer 是一个含有 3 个整数元素的列表，它代表循环缓冲区；occupiedBufferCount 指出存在多少个被占用的缓冲区；readLocation 指出消费者读取下一个值的位置；writeLocation 指出生产者写入下一个值的位置；而 threadCondition 是一个条件变量，用于对缓冲区访问进行保护。

set 方法（第 20~50 行）执行的任务和图 19.8 相同，只不过略有更改。while 结构（第 27~30 行）判断生产者是否必须等待（即是否所有缓冲区为满）。如果生产者线程必须等待，第 28~29 行就打印一条消息，指出生产者正在等着执行任务。而第 30 行调用 Condition 的 wait 方法。从 while 结构之后的第 36 行继续执行时，由生产者写入的值会放到循环缓冲区中 writeLocation 所指定的位置处。接着，第 37~38 行打印一条消息，其中包含产生的值。第 41 行使 occupiedBufferCount 值自增，因为缓冲区现在至少包含一个可供消费者读取的值。接着，第 45~46 行更新 writeLocation，为下一次 set 方法调用做好准备。第 47 行调用 CircularBuffer 的 displayState 方法（第 86~117 行），以便创建输出，指出已经占用的缓冲区的数目、缓冲区的内容以及当前的 writeLocation 和 readLocation 值。然后，调用条件变量的 notify 方法，要求正在等待的方法（如果有的话）转变成就绪状态。最后调用条件变量的 release 方法，释放条件变量的基本锁，允许另一个线程获得该锁以访问共享缓冲区。

本例的 `get` 方法（第 52~84 行）执行的任务和在图 19.8 是相同的，只不过略有更改。第 59~62 行的 `while` 结构判断消费者是否必须等待（即是否所有缓冲区为空）。如果是，第 60~61 行打印一条消息，指出消费者正在等着执行任务，而且第 62 行调用 `Condition` 的 `wait` 方法。当执行最终从 `while` 结构之后的第 68 行继续时，会将循环缓冲中位于 `readLocation` 位置的值指派给 `tempNumber`。第 69~70 行打印一条消息，其中包含已经使用的值。第 73 行使 `occupiedBufferCount` 值自减，因为缓冲区至少包含一个开放的位置，生产者可在其中放入一个值。之后，第 77~78 行更新 `readLocation`，为下一次调用 `get` 方法做好准备。然后，第 79 行调用 `displayState` 方法来创建输出，指出被占用的缓冲区的个数、缓冲区的内容以及当前的 `writeLocation` 和 `readLocation` 值。接着，调用条件变量的 `notify` 方法，要求正在等待的线程（如果有的话）应该转变成“ready”状态。最后调用条件变量的 `release` 方法，释放条件变量的基本锁。第 84 行将取得的值返回调用线程。

19.9 信号机

“信号机”（Semaphore）是一个变量，控制着对公共资源或临界区的访问。信号机维护着一个计数器，指定可同时使用资源或进入临界区的线程数。每次有一个线程获得信号机时，计数器都会自减。若计数器为零，其他线程便只能暂停访问信号机，直到另一个线程释放信号机。图 19.14 以一个饭馆为例，演示如何用信号机控制对临界区的访问。

```

1 # Figure 19.14: fig19_14.py
2 # Semaphore to control access to a critical section.
3
4 import threading
5 import random
6 import time
7
8 class SemaphoreThread( threading.Thread ):
9     """Class using semaphores"""
10
11     availableTables = [ "A", "B", "C", "D", "E" ]
12
13     def __init__( self, threadName, semaphore ):
14         """Initialize thread"""
15
16         threading.Thread.__init__( self, name = threadName )
17         self.sleepTime = random.randrange( 1, 6 )
18
19         # set the semaphore as a data attribute of the class
20         self.threadSemaphore = semaphore
21
22     def run( self ):
23         """Print message and release semaphore"""
24
25         # acquire the semaphore
26         self.threadSemaphore.acquire()
27
28         # remove a table from the list
29         table = SemaphoreThread.availableTables.pop()
30         print "%s entered; seated at table %s." % \
31             ( self.getName(), table ),
32         print SemaphoreThread.availableTables
33
34         time.sleep( self.sleepTime ) # enjoy a meal
35
36         # free a table
37         print " %s exiting; freeing table %s." % \
38             ( self.getName(), table ),
39         SemaphoreThread.availableTables.append( table )
40         print SemaphoreThread.availableTables
41
42         # release the semaphore after execution finishes

```

```

43     self.threadSemaphore.release()
44
45     threads = [] # list of threads
46
47     # semaphore allows five threads to enter critical section
48     threadSemaphore = threading.Semaphore(
49         len( SemaphoreThread.availableTables ) )
50
51     # create ten threads
52     for i in range( 1, 11 ):
53         threads.append( SemaphoreThread( "thread" + str( i ),
54             threadSemaphore ) )
55
56     # start each thread
57     for thread in threads:
58         thread.start()

```

```

thread1 entered; seated at table E. ['A', 'B', 'C', 'D']
thread2 entered; seated at table D. ['A', 'B', 'C']
thread3 entered; seated at table C. ['A', 'B']
thread4 entered; seated at table B. ['A']
thread5 entered; seated at table A. []
  thread2 exiting; freeing table D. ['D']
thread6 entered; seated at table D. []
  thread1 exiting; freeing table E. ['E']
thread7 entered; seated at table E. []
  thread3 exiting; freeing table C. ['C']
thread8 entered; seated at table C. []
  thread4 exiting; freeing table B. ['B']
thread9 entered; seated at table B. []
  thread5 exiting; freeing table A. ['A']
thread10 entered; seated at table A. []
  thread7 exiting; freeing table E. ['E']
  thread8 exiting; freeing table C. ['E', 'C']
  thread9 exiting; freeing table B. ['E', 'C', 'B']
  thread10 exiting; freeing table A. ['E', 'C', 'B', 'A']
  thread6 exiting; freeing table D. ['E', 'C', 'B', 'A', 'D']

```

图 19.14 用信号机控制对临界区的访问

第 48~49 行创建一个 `threading.Semaphore` 对象，它最多允许 5 个线程访问临界区。`Semaphore` 类的一个对象用计数器跟踪获取和释放信号机的线程数量。

第 52~54 行创建一个列表，该列表由 `SemaphoreThread` 对象构成。`start` 方法开始列表中的每个线程（第 57~58 行）。

`SemaphoreThread` 类（第 8~43 行）的每个对象代表饭馆里的一个客人。类属性 `availableTables`（第 11 行）跟踪饭馆中可用的桌子。

信号机有一个内建计数器，用于跟踪它的 `acquire` 和 `release` 方法调用的次数。内部计数器的初始值可作为参数传给 `Semaphore` 构造函数。默认值为 1。计数器大于 0，`Semaphore` 的 `acquire` 方法（第 26 行）就为线程获得信号机，并使计数器自减。如计数器为零，线程会暂停，直到另一个线程释放了信号机。信号机的计数器永远不会小于 0。

线程开始执行 `SemaphoreThread` 类的 `run` 方法中的临界区时，列表方法 `pop`（第 29 行）会从 `availableTables` 移除最后一项，并把它指派给 `table` 变量。程序显示哪个线程进入临界区，以及那个线程所代表的客人坐在哪张桌子旁。然后，线程休眠（占据一张桌子）随机时间（第 34 行），该时间是在 `SemaphoreThread` 对象构造时计算好的。线程准备退出临界区时，第 39 行将 `table` 的值追加到 `availableTables` 后，并显示最新的可用餐桌列表。

`Semaphore` 的 `release` 方法（第 43 行）在线程结束临界区的执行后释放信号机。方法调用使 `Semaphore` 的计数器自增，并通知一个正在等待的线程（如果有的话）进入“ready”状态，以等待执行。

注意在图 19.14 中，如果移除第 26 行和第 43 行，可能会有 5 个以上的线程试图从共享列表移除一个项目，导致 `IndexError` 异常。

19.10 事件

threading 模块定义了可用于线程通信的 Event (事件) 类。Event 有一个内部标志, 可为 true 或 false。一个或多个线程能调用 Event 对象的 wait 方法以暂停并等待事件发生。事件发生后, 暂停的线程 (可能有多) 会按它们抵达的顺序被唤醒, 并恢复执行。图 19.15 演示了红绿灯每隔 3 秒钟变绿的情况。我们用一个 Event 对象 (而不是使用条件变量或其他同步机制) 来同步线程, 因为线程并不共享任何数据。相反, 线程只需在灯变绿时接收一个通知。

第 36 行创建一个 Event 对象, 名为 greenLight, 以模拟红绿灯。第 37~42 行创建一个由 VehicleThread 构成的列表。VehicleThread 类 (第 8~34 行) 代表在十字路口的一辆车。第 44~45 行开始车辆线程。每个线程都休眠随机时间, 打印一条到达消息, 等待绿灯 (即 greenLight 的内部标志为 true), 并打印一条离开消息。

while 结构 (第 47~57 行) 将一直循环, 直到只有主线程保持活动 (也就是说, 在所有车辆线程终止以后)。每次循环时, 都会调用 Event 的 clear 方法, 休眠 3 秒钟, 调用 Event 的 set 方法, 并休眠 1 秒钟。Event 的 clear 和 set 方法分别将内部标志值变成 false 和 true。注意, set 方法 (第 56 行) 按车辆线程抵达的顺序依次唤醒它们。之后, 每个线程都可打印自己正在通过十字路口的消息。

```

1 # Fig. 19.15: fig19_15.py
2 # Event objects.
3
4 import threading
5 import random
6 import time
7
8 class VehicleThread( threading.Thread ):
9     """Class representing a motor vehicle at an intersection"""
10
11     def __init__( self, threadName, event ):
12         """Initializes thread"""
13
14         threading.Thread.__init__( self, name = threadName )
15
16         # ensures that each vehicle waits for a green light
17         self.threadEvent = event
18
19     def run( self ):
20         """Vehicle waits unless/until light is green"""
21
22         # stagger arrival times
23         time.sleep( random.randrange( 1, 10 ) )
24
25         # prints arrival time of car at intersection
26         print "%s arrived at %s" % \
27             ( self.getName(), time.ctime( time.time() ) )
28
29         # flag is false until light is green
30         self.threadEvent.wait()
31
32         # displays time that car departs intersection
33         print "%s passes through intersection at %s" % \
34             ( self.getName(), time.ctime( time.time() ) )
35
36 greenLight = threading.Event()
37 vehicleThreads = []
38
39 # creates and starts ten Vehicle threads
40 for i in range( 1, 11 ):
41     vehicleThreads.append( VehicleThread( "Vehicle" + str( i ),
42         greenLight ) )
43
44 for vehicle in vehicleThreads:
45     vehicle.start()

```



```

46
47 while threading.activeCount() > 1:
48
49     # sets the Event's flag to false -- block all incoming vehicles
50     greenLight.clear()
51     print "RED LIGHT! at", time.ctime( time.time() )
52     time.sleep( 3 )
53
54     # sets the Event's flag to true -- awaken all waiting vehicles
55     print "GREEN LIGHT! at", time.ctime( time.time() )
56     greenLight.set()
57     time.sleep( 1 )

```

```

RED LIGHT! at Fri Dec 21 18:10:44 2001
Vehicle7 arrived at Fri Dec 21 18:10:45 2001
Vehicle2 arrived at Fri Dec 21 18:10:46 2001
Vehicle8 arrived at Fri Dec 21 18:10:46 2001
GREEN LIGHT! at Fri Dec 21 18:10:47 2001
Vehicle7 passes through intersection at Fri Dec 21 18:10:47 2001
Vehicle2 passes through intersection at Fri Dec 21 18:10:47 2001
Vehicle8 passes through intersection at Fri Dec 21 18:10:47 2001
Vehicle1 arrived at Fri Dec 21 18:10:48 2001
Vehicle1 passes through intersection at Fri Dec 21 18:10:48 2001
Vehicle9 arrived at Fri Dec 21 18:10:48 2001
Vehicle9 passes through intersection at Fri Dec 21 18:10:48 2001
Vehicle10 arrived at Fri Dec 21 18:10:48 2001
Vehicle10 passes through intersection at Fri Dec 21 18:10:48 2001
RED LIGHT! at Fri Dec 21 18:10:48 2001
Vehicle4 arrived at Fri Dec 21 18:10:50 2001
Vehicle5 arrived at Fri Dec 21 18:10:50 2001
Vehicle6 arrived at Fri Dec 21 18:10:51 2001
GREEN LIGHT! at Fri Dec 21 18:10:51 2001
Vehicle4 passes through intersection at Fri Dec 21 18:10:51 2001
Vehicle5 passes through intersection at Fri Dec 21 18:10:51 2001
Vehicle6 passes through intersection at Fri Dec 21 18:10:51 2001
RED LIGHT! at Fri Dec 21 18:10:52 2001
Vehicle3 arrived at Fri Dec 21 18:10:53 2001
GREEN LIGHT! at Fri Dec 21 18:10:55 2001
Vehicle3 passes through intersection at Fri Dec 21 18:10:55 2001

```

图 19.15 演示 Event 对象的红绿灯示例

本章介绍了多线程处理和同步机制。我们用简单的例子演示了程序如何访问这些类，以创建线程和维持数据完整性。下一章将利用这些技术创建 Tic-Tac-Toe 联网游戏。

第 20 章 联 网

学习目标

- 理解 Python 联网元素，即 URL、套接字和数据文报
- 使用套接字和数据文报实现 Python 联网应用程序
- 理解如何用 Python 实现客户/服务器应用程序
- 理解如何创建基于网络的协作式应用程序
- 构建一个多线程服务器

20.1 概述

Internet 和 Web 极大促进了商业和计算领域的发展。Internet 将“信息世界”紧紧连到一起；Web 则使 Internet 更易使用，同时还增加了多媒体支持。许多组织都将 Internet 和 Web 视为自己的信息系统策略的关键组件。Python 提供许多内建的联网能力，它们可简化基于 Internet 和 Web 的应用程序开发。Python 不仅通过多线程处理实现了并发性，还允许程序通过 Web 请求信息，并与远程计算机上运行的程序协作。

第 6 章展示了 Python 的联网和分布式计算能力。我们讨论了服务器端 Web 技术，它允许用户为网站生成动态 Web 内容。本章讨论的联网将重点同时放在客户/服务器关系的两端。客户请求服务器采取某项行动；服务器采取行动以响应客户。这种“请求 - 响应”模型的最常见的一种实现就是 Web 浏览器和 Web 服务器。用户通过浏览器（客户应用程序）选择一个要查看的网站时，浏览器向相应的 Web 服务器（服务器应用程序）发出一个请求。服务器通常发送相应的网页，以便响应客户。

本章的重点是如何利用 Python 的联网能力构建分布式应用程序。本章将介绍 Python 的基于套接字的通信，它允许应用程序将联网视为文件 I/O（程序可从套接字获取数据，也可向套接字发送数据，就像平时读写文件那样）。我们还将展示如何创建和操纵套接字。Python 提供了流套接字和数据文报套接字。通过流套接字，进程要建立和另一个进程的连接。如果有连接，数据就会以连续的数据流的形式在进程之间流动。流套接字提供了面向连接的服务。数据传输协议是流行的 TCP（传输控制协议），第 6 章已对其进行了介绍。

UDP（用户数据文报协议）允许通过数据文报套接字进行无连接的网络通信。对于普通应用，这并不是一个高效的协议，因为 UDP 是一种无连接的服务，不能保证数据包按特定的顺序抵达。事实上，数据包除了可能按混乱的顺序抵达之外，还可能丢失或者重复。所以，假如使用 UDP，就可能需要在用户端进行附加的编程，以解决这些问题。对于大多数 Python 程序员，流套接字和 TCP 协议仍然是他们最理想的选择。

20.2 通过 HTTP 定址 URL

Internet 使用了大量协议，有的已在第 6 章讲解。本节要复习某些 Web 协议，并在一个例子中使用 Python 模块来显示网页。最重要的协议之一是“超文本传输协议”（HTTP），它是在 Web 上进行数据传输的关键。HTTP 使用 URL（统一资源定位符）来定位 Internet 上的内容。URL 对应于文件、目录或复杂任务（比如数据库查询和 Internet 搜索）。

图 20.1 使用 Tkinter 和 Pmw 这两个 GUI 组件在 Web 浏览器中显示 Web 服务器上的一个文件的内容。为此，我们首先定义 WebBrowser 类，它相当于一个 Web 浏览器。用户在浏览器顶部的文本字段中输入 URL，相应的 Web 文档（如果有的话）则显示在下方的滚动文本区域中。

WebBrowser 类中包含名为 address 的 Entry 组件，可在其中输入要读取的文件的 URL。一个名为

contents 的 ScrolledText 组件用于显示文件内容。用户在 Entry 组件中按回车键后，会执行 getPage 方法。getPage 方法（第 30~50 行）从 Web 服务器获取指定的文件。第 34 行调用 address 组件的 get 方法，从而获得 URL。

urlparse 模块提供了用于解析 URL 的函数，以及用于 URL 处理的其他函数。urlparse.urlparse 函数取得一个字符串作为输入，并返回一个含有 6 个元素的元组。元组的第一个元素是定址方案，本例使用的是 http。输入以 http 开头的 URL，Web 服务器就会获取并传输与请求的 URL 对应的文档。第 39 行检查用户输入的 URL 是否以“http://”开头。如果不是，就假定用户忘记添加“http://”，并自动把它添加到 URL 之前（第 40 行）。

```

1 # Fig. 20.1: fig20_01.py
2 # Displays the contents of a file from a Web server in a browser.
3
4 from Tkinter import *
5 import Pmw
6 import urllib
7 import urlparse
8
9 class WebBrowser( Frame ):
10     """A simple Web browser"""
11
12     def __init__( self ):
13         """Create the Web browser GUI"""
14
15         Frame.__init__( self )
16         Pmw.initialise()
17         self.pack( expand = YES, fill = BOTH )
18         self.master.title( "Simple Web Browser" )
19         self.master.geometry( "400x300" )
20
21         self.address = Entry( self )
22         self.address.pack( fill = X, padx = 5, pady = 5 )
23         self.address.bind( "<Return>", self.getPage )
24
25         self.contents = Pmw.ScrolledText( self,
26             text_state = DISABLED )
27         self.contents.pack( expand = YES, fill = BOTH, padx = 5,
28             pady = 5 )
29
30     def getPage( self, event ):
31         """Parse URL, add addressing scheme and retrieve file"""
32
33         # parse the URL
34         myURL = event.widget.get()
35         components = urlparse.urlparse( myURL )
36         self.contents.text_state = NORMAL
37
38         # if addressing scheme not specified, use http
39         if components[ 0 ] == "":
40             myURL = "http://" + myURL
41
42         # connect and retrieve the file
43         try:
44             tempFile = urllib.urlopen( myURL )
45             self.contents.settext( tempFile.read() ) # show results
46             tempFile.close()
47         except IOError:
48             self.contents.settext( "Error finding file" )
49
50         self.contents.text_state = DISABLED
51
52     def main():
53         WebBrowser().mainloop()
54
55 if __name__ == "__main__":
56     main()

```

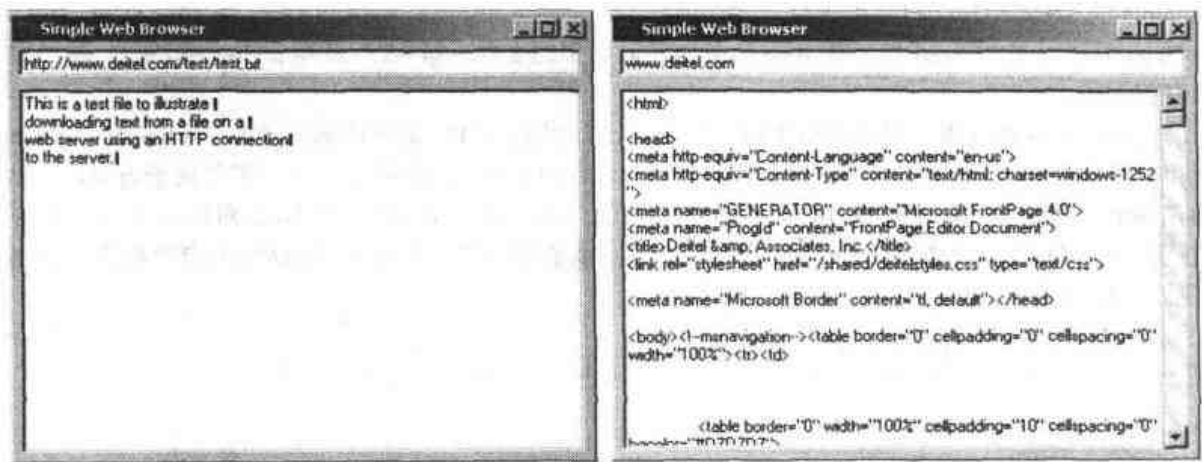


图 20.1 用于读取文件的 URL 连接

第 43~48 行试图连接 Web 服务器，并使用 `urllib` 模块来获取请求的文件。`urllib` 模块提供的方法可访问一个 URL 所引用的数据。第 45 行将 URL 传给 `urllib` 的 `urlopen` 函数，以便获取文件。该函数会导致执行一次 DNS（域名系统）查询。DNS 服务器将域名（或 URL）转换成 IP 地址，该地址惟一性地标识了网络上的一台计算机。模块从 Web 服务器请求文档。如果成功，`urlopen` 就返回一个对象。该对象的行为就像一个 Python 文件对象，而且它还可以使用文件方法，比如 `read`、`readline`、`readlines` 和 `close`。第 45 行读取文件并在 `contents` 组件中显示结果。然后，第 46 行关闭文件。如果 `urlopen` 失败，第 48 行将向用户显示一条错误消息。

20.3 建立简单服务器（使用流套接字）

典型情况下，使用 TCP 和流套接字，服务器要“等待”来自客户的一个连接请求。通常，服务器端应用程序包含一个控制结构或代码块，它将连续执行，直到服务器收到一个连接请求。收到请求后，服务器建立和客户的连接。然后，服务器用这个连接处理来自那个客户的请求，并将数据发送给客户。

要在 Python 中建立具有 TCP 和流套接字的简单服务器，需要使用 `socket` 模块。利用该模块包含的函数和类定义，可生成通过网络通信的程序。建立这个连接需要 6 个步骤。

第 1 步是创建 `socket` 对象。调用 `socket` 构造函数，例如：

```
socket = socket.socket( family, type )
```

就可使用指定的地址家族（*family*）和类型（*type*）创建一个套接字。通常，*family* 参数可为 `AF_INET` 或 `AF_UNIX`。该参数指定如何解释套接字使用的地址。`AF_INET` 家族包括 Internet 地址，`AF_UNIX` 家族用于同一台机器上的进程间通信。本章只用 `AF_INET`。至于 *type* 参数，它典型的值是 `SOCK_STREAM`（流套接字）和 `SOCK_DGRAM`（数据文报套接字）。这些常量是在 `socket` 模块中定义的。为方便讨论，我们假定已创建了一个流套接字。20.6 节要讨论另一种套接字类型——数据文报套接字。

创建服务器的第 2 步是将 `socket` 绑定（指派）到指定地址。这是通过调用 `socket` 对象的 `bind` 方法来实现的，例如：

```
socket.bind( address )
```

对于 `AF_INET` 家族所创建的一个套接字，*address*（地址）必须是一个双元素元组，形如（*host*, *port*）。其中，*host*（主机）是一个代表主机名或 IP 地址的字符串，*port* 则是端口号（整数）。上述语句将保留一个端口，服务器在此等待来自客户的连接。客户则通过该端口连接服务器。如果端口正在使用、主机名不

正确或者端口已被保留, `bind` 方法将引发 `socket.error` 异常。

软件工程知识 20.1 端口号范围在 0~65 535 之间。很多操作系统为系统服务(比如电子邮件和 Web 服务器)保留了 1024 以下的端口号。应用程序在得到特别授权后才能使用这些保留端口号。服务器端应用程序一般不要把 1024 以下的端口号指定为连接端口。

常见编程错误 20.1 创建 `socket` 时, 试图指定已分配的或者无效的端口会引发异常。

套接字绑定到一个地址后, 必须准备好套接字以便接收连接请求(第 3 步)。为此, 需要使用 `socket` 的 `listen` 方法:

```
socket.listen( backlog )
```

其中, `backlog` 指定了最多允许多少个客户连接到服务器。它的值至少为 1。收到连接请求后, 这些请求需要排队。如果队列是满的, 就拒绝请求。

在第 4 步, 服务器套接字通过 `socket` 的 `accept` 方法等待客户请求一个连接:

```
connection, address = socket.accept()
```

调用 `accept` 方法时, `socket` 会进入“waiting”(或阻塞)状态。客户请求连接时, 方法建立连接并返回服务器。`accept` 方法返回一个含有两个元素的元组, 形如(`connection`, `address`)。第一个元素(`connection`)是新的 `socket` 对象, 服务器通过它与客户通信; 第二个元素(`address`)是客户的 Internet 地址。

第 5 步是处理阶段, 服务器和客户通过 `send` 和 `recv` 方法通信(传输数据)。服务器调用 `send`, 并采用字符串形式向客户发送信息。`send` 方法返回已发送的字符个数。服务器使用 `recv` 方法从客户接收信息。调用 `recv` 时, 服务器必须指定一个整数, 它对应于可通过本次方法调用来接收的最大数据量。`recv` 方法在接收数据时会进入“blocked”状态, 最后返回一个字符串, 用它来表示收到的数据。如果发送的数据量超过了 `recv` 所允许的, 数据会被截短。多余的数据将缓冲于接收端。以后调用 `recv` 时, 多余的数据会从缓冲区删除(以及自从上次调用 `recv` 以来, 客户可能发送的其他任何数据)。

在第 6 步, 传输结束, 服务器调用 `socket` 的 `close` 方法以关闭连接。

常见编程错误 20.2 套接字 `send` 方法只接受一个字符串参数。传递不同类型的值(比如整数)会出错。

软件工程知识 20.2 利用 Python 的多线程能力, 程序员可创建多线程服务器, 以便同时管理多个并发的客户连接。

软件工程知识 20.3 多线程服务器可用每个 `accept` 调用返回的套接字来创建一个线程, 由它来管理通过该套接字进行的网络 I/O。另外, 可让多线程服务器维护一个线程池, 以便管理通过新建套接字进行的网络 I/O。

性能提示 20.1 在内存充足的高性能系统中, 多线程服务器可创建一个线程池。可快速分配这些线程, 以管理通过每个新建套接字进行的网络 I/O。以后收到连接请求时, 服务器不会产生创建线程的开销。

20.4 建立简单客户(使用流套接字)

在 Python 中建立一个简单客户需要 4 个步骤。第 1 步是创建一个 `socket` 以连接服务器:

```
socket = socket.socket( family, type )
```

第 2 步是使用 `socket` 的 `connect` 方法连接服务器。`connect` 方法的参数是准备连接的套接字的地址。对于 `AF_INET` 客户套接字, `connect` 方法调用的形式是:

```
socket.connect( ( host, port ) )
```

其中, *host* 是一个代表服务器主机名或 IP 地址的字符串, *port* 是服务器进程所绑定的整数端口号。如连接成功, 客户可通过套接字与服务器通信。连接失败, 会引发 `socket.error` 异常。

常见编程错误 20.3 客户指定的服务器地址不能解析或连接服务器时出错, 都会引发 `socket.error` 异常。

第 3 步是处理阶段, 客户和服务器将通过 `send` 和 `recv` 方法通信。在第 4 步, 当传输结束后, 客户通过调用 `socket` 的 `close` 方法来关闭连接。

20.5 通过流套接字连接进行客户/服务器交互

图 20.2 和图 20.3 的程序使用流套接字和前两节介绍的技术来构建一个简单的客户/服务器聊天程序。服务器等待客户的连接请求。客户应用程序连接服务器时, 服务器应用程序向客户发送一个字符串, 表示连接成功。然后, 客户显示一条消息, 表示已经建立连接。

客户和服务器应用程序都允许用户输入一条消息, 并把它发送给其他应用程序。客户或服务器发送字符串 "TERMINATE" 后, 客户和服务器的连接会终止。然后, 服务器等待另一个客户请求连接。图 20.2 和图 20.3 分别包含了服务器和客户定义。图 20.3 还包含示范输出, 演示了两者进行交互的结果。

```
1 # Fig. 20.2: fig20_02.py
2 # Set up a server that will receive a connection
3 # from a client, send a string to the client,
4 # and close the connection.
5
6 import socket
7 import sys
8
9 HOST = "127.0.0.1"
10 PORT = 5000
11 counter = 0
12
13 # step 1: create socket
14 mySocket = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
15
16 # step 2: bind the socket to address
17 try:
18     mySocket.bind( ( HOST, PORT ) )
19 except socket.error:
20     sys.exit( "Call to bind failed" )
21
22 while 1:
23
24     # step 3: wait for connection request
25     print "Waiting for connection"
26     mySocket.listen( 1 )
27
28     # step 4: establish connection for request
29     connection, address = mySocket.accept()
30     counter += 1
31     print "Connection", counter, "received from:", address[0]
32
33     # step 5: send and receive data via connection
34     connection.send( "SERVER>>> Connection successful" )
35     clientMessage = connection.recv( 2024 )
36
37     while clientMessage != "CLIENT>>> TERMINATE":
38
39         if not clientMessage:
40             break
41
42         print clientMessage
43         serverMessage = raw_input( "SERVER>>> " )
```

```

44     connection.send( "SERVER>>> " + serverMessage )
45     clientMessage = connection.recv( 1024 )
46
47     # step 6: close connection
48     print "Connection terminated"
49     connection.close()

```

图 20.2 使用面向连接的传输并通过套接字传输数据 - 服务器端应用程序

```

1  # Fig. 20.3: fig20_03.py
2  # Set up a client that will read information sent
3  # from a server and display that information.
4
5  import socket
6  import sys
7
8  HOST = "127.0.0.1"
9  PORT = 5000
10
11 # step 1: create socket
12 print "Attempting connection"
13 mySocket = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
14
15 # step 2: make connection request to server
16 try:
17     mySocket.connect( ( HOST, PORT ) )
18 except socket.error:
19     sys.exit( "Call to connect failed" )
20
21 print "Connected to Server"
22
23 # step 3: transmit data via connection
24 serverMessage = mySocket.recv( 1024 )
25
26 while serverMessage != "SERVER>>> TERMINATE":
27
28     if not serverMessage:
29         break
30
31     print serverMessage
32     clientMessage = raw_input( "CLIENT>>> " )
33     mySocket.send( "CLIENT>>> " + clientMessage )
34     serverMessage = mySocket.recv( 1024 )
35
36 # step 4: close connection
37 print "Connection terminated"
38 mySocket.close()

```

```

Waiting for connection
Connection 1 received from: 127.0.0.1

```

```

Attempting connection
Connected to Server
SERVER>>> Connection successful
CLIENT>>> Hi to person at server

```

```

Waiting for connection
Connection 1 received from: 127.0.0.1
CLIENT>>> Hi to person at server
SERVER>>> Hi back to you--client!

```

```

Attempting connection
Connected to Server
SERVER>>> Connection successful
CLIENT>>> Hi to person at server
SERVER>>> Hi back to you--client!
CLIENT>>> TERMINATE

```

```

Waiting for connection

```

```

Connection 1 received from: 127.0.0.1
CLIENT>>> Hi to person at server
SERVER>>> Hi back to you--client!
Connection terminated
Waiting for connection

```

图 20.3 使用面向连接的传输并通过套接字传输数据 - 客户端应用程序

第 14~45 行允许服务器等待一个连接请求，建立一个连接，并通过该连接传输数据。第 14 行创建了套接字对象 `mySocket` 以等待连接请求。整数 `counter` 代表已建立的连接总数（第 11 行）。

第 18 行将 `mySocket` 绑定到端口 5000。如果发生套接字错误，程序将打印一条错误消息（第 20 行）。注意，`HOST` 是字符串“127.0.0.1”。这个 IP 地址引用的肯定是本机（也称为 `localhost`）。注意，为演示客户/服务器关系，我们选择在同一台机器（即 `localhost`）上执行的程序之间建立连接。要用本例进行真正的 Internet 客户/服务器交互，第一个参数通常是包含另一台计算机的 Internet 地址的字符串。第 22~45 行包含一个 `while` 循环，服务器在其中不断地侦听每个客户请求，并在收到请求后建立连接。第 26 行在端口 5000 侦测来自客户的连接请求。`listen` 方法的参数是在队列中等待连接服务器的请求的数量（本例是 1）。客户请求连接时如果队列是满的，服务器会拒绝建立连接。

`listen` 方法会设置一个侦听器，以等待客户请求连接。收到请求后，`socket` 的 `accept` 方法（第 29 行）会创建一个 `socket` 对象以管理连接。记住，`accept` 返回的是一个双元素元组。第一个元素是新的 `socket` 对象，我们把它命名为 `connect`；第二个元素是连接该服务器的客户机的 Internet 地址——对于 `AF_INET` 套接字，采取的是 `(host, port)` 形式。针对当前连接存在一个新的 `socket`，第 31 行会打印一条消息，显示连接编号和客户地址。

第 34 行调用 `socket` 的 `send` 方法，将字符串“SERVER>>> Connection successful”发送给客户。第 35 行调用 `socket` 的 `recv` 方法从客户接收一个字符串，其中最多可包含 1024 个字节。第 37~45 行的 `while` 循环会不断执行，直到服务器收到消息“CLIENT>>> TERMINATE”。第 39~40 行判断连接是否由客户关闭。一旦连接关闭，`recv` 就返回一个空字符串。在这种情况下，`break` 语句将退出循环；否则，第 42 行将打印从客户收到的消息。

`raw_input` 函数（第 43 行）从用户读取一个字符串。服务器将这个字符串发送给客户（第 44 行），并从客户接收一条消息（第 45 行）。传输结束后，第 49 行就关闭套接字。服务器将等待来自客户的下一个请求。

本例采取的思路是：服务器接收连接请求，建立连接并通过它传输数据，关闭连接并等待下一个请求。但更好的思路是：服务器接收一个请求，设置连接以便由一个单独的线程管理，再等待其他请求。对连接进行处理的线程在执行时，服务器可继续处理连接请求。这样一来，多线程服务器应用程序的效率会更高。

图 20.3 显示了客户程序及其与服务器的交互（通过一个流套接字连接进行）。另外也显示了一些示范输出。

第 13~34 行将客户连接到服务器，允许客户从服务器接收数据，并允许客户将数据发送给服务器。第 13 行创建 `socket` 对象 `mySocket` 以建立一个连接。然后，客户调用 `socket` 的 `connect` 方法请求连接服务器。该方法的参数是一个双元素元组（第 17 行）。和图 20.2 一样，将 `PORT` 变量设为一样的值（5000），可保证客户 `socket` 通过与服务端一样的端口来请求连接。如果 `connect` 调用引发 `socket.error` 异常，`except` 语句就打印错误消息。

如连接成功，第 21 行会打印一条消息。`socket` 的 `recv` 方法（第 24 行）从服务器接收一条消息（即“SERVER>>> Connection successful”）。`while` 循环（第 26~34 行）会一直执行，直到客户收到消息“SERVER>>> TERMINATE”。和服务端程序一样，第 28 行检查收到的每一条消息，判断服务器是否关闭了连接。如果是，`break` 语句就退出 `while` 循环（第 29 行）。

每次循环，都要打印服务器所发送的消息（第 31 行），并调用 `raw_input` 函数从用户那里读取一个字符串（第 32 行）。第 33 行调用 `socket` 的 `send` 方法，将这个字符串发送给服务器。然后，客户接收来

自服务器的下一条消息（第 34 行）。传输结束后，第 38 行关闭 socket 对象 mySocket。

图 20.2 和图 20.3 的程序提供了基本的客户/服务器通信功能。更高级的功能需要更复杂的代码。例如，上述程序无法同时处理来自客户的多条消息，也不能处理长度超过 1024 个字节的消息。编写专业联网应用程序时，这些问题会使开发愈加复杂化。

20.6 通过数据文报进行无连接的客户/服务器交互

迄今为止，我们讨论的只是“面向连接的、基于流的”传输，下面将接着讨论如何使用数据文报进行“无连接的”传输。

面向连接的传输类似于通过电话系统交互。用户首先拨号，并连接到目标用户的电话机。打电话时，系统会维持这个连接，不管用户是否交谈。

相反，通过数据文报进行的无连接传输更接近传统的邮政服务。它首先以数据包（名为“数据文报”）的形式捆绑和发送信息，可将这些信息想象成邮政信件。一封较大的信件在一个信封中装不下，就分解成多个信件片段，并放到单独的、顺序编号的信封中。所有信件同时发出。这些信件可能按正确顺序抵达，可能按混乱的顺序抵达，或者根本没有抵达（虽然最后一种情况较为少见，但确有可能发生）。位于接收端的人先将这些散落的片段组合成正确的顺序，然后才能解释消息。如果信件本来就非常小，一个信封便足够了，就不会出现排序问题。但即使在这种情况下，仍有可能出现信件根本没有抵达的情况。数据文报和邮政服务的一个区别在于，可能会有重复的数据文报抵达接收计算机。

图 20.4 和图 20.5 的程序使用数据文报在客户和服务器应用程序之间发送数据包（即含有“信件”的“信封”）。在客户应用程序中，用户输入消息并按回车。客户将消息放到一个数据文报数据包中，并发送给服务器。服务器接收这个数据包，显示其中的信息，再将数据包回送（或返回）给客户。客户收到数据包后，将显示其中的信息。

图 20.4 的服务器代码定义了一个 socket 对象，该对象负责发送和接收数据文报（SOCK_DGRAM）数据包。注意 socket 的类型设为 SOCK_DGRAM，这表明 mySocket 是一个用数据文报传输数据的套接字。第 14 行将 socket 绑定到用于从客户那里接收数据包的端口（5000），客户也用这个端口向服务器发送数据包。

```

1 # Fig. 20.4: fig20_04.py
2 # Set up a server that will receive packets from a
3 # client and send packets to a client.
4
5 import socket
6
7 HOST = "127.0.0.1"
8 PORT = 5000
9
10 # step 1: create socket
11 mySocket = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
12
13 # step 2: bind socket
14 mySocket.bind( ( HOST, PORT ) )
15
16 while 1:
17
18     # step 3: receive packet
19     packet, address = mySocket.recvfrom( 1024 )
20
21     print "Packet received:"
22     print "From host:", address[ 0 ]
23     print "Host port:", address[ 1 ]
24     print "Length:", len( packet )
25     print "Containing:"
26     print "\t" + packet
27
28     # step 4: echo packet back to client

```

```

29     print "\nEcho data to client...",
30     mySocket.sendto( packet, address )
31     print "Packet sent\n"
32
33 mySocket.close()

```

```

Packet received:
From host: 127.0.0.1
Host port: 1645
Length: 20
Containing:
    first message packet

Echo data to client... Packet sent

```

图 20.4 使用数据文报无连接地传输数据 - 服务器端应用程序

第 16~31 行的 while 循环从客户那里接收数据包。首先，第 19 行等待一个数据包抵达。recvfrom 方法会阻塞（暂停），直到数据包抵达。之后，recvfrom 返回一个字符串，表示已收到的数据以及发送数据的套接字的地址。然后，服务器打印一条消息，其中包含客户地址以及客户发送的数据。第 30 行调用 socket 的 sendto 方法，将数据回送给客户。方法的第一个参数是 packet，它指定了要发送的数据。第二个参数是 address 元组，指定数据包要发送到的客户机的 Internet 地址以及服务器用于接收数据包的端口。

客户代码（图 20.5）与服务器代码相似；但是，只有在用户输入消息并按回车键之后，客户才会发送数据包。while 循环（第 13~29 行）使用 sendto 方法（第 18 行）将数据包发送给服务器，并用 recvfrom 方法（第 22 行）等待数据包。recvfrom 方法会暂时阻塞，直到有一个数据包抵达。

```

1 # Fig. 20.5: fig20_05.py
2 # Set up a client that will send packets to a
3 # server and receive packets from a server.
4
5 import socket
6
7 HOST = "127.0.0.1"
8 PORT = 5000
9
10 # step 1: create socket
11 mySocket = socket.socket( socket.AF_INET, socket.SOCK_DGRAM )
12
13 while 1:
14
15     # step 2: send packet
16     packet = raw_input( "Packet>>>" )
17     print "\nsending packet containing:", packet
18     mySocket.sendto( packet, ( HOST, PORT ) )
19     print "Packet sent\n"
20
21     # step 3: receive packet back from server
22     packet, address = mySocket.recvfrom( 1024 )
23
24     print "Packet received:"
25     print "From host:", address[ 0 ]
26     print "Host port:", address[ 1 ]
27     print "Length:", len( packet )
28     print "Containing:"
29     print "\t" + packet + "\n"
30
31 mySocket.close()

```

```

Packet>>>first message packet

Sending packet containing: first message packet
Packet sent

Packet received:
From host: 127.0.0.1
Host port: 5000

```

```

Length: 20
Containing:
    first message packet
Packet>>>

```

图 20.5 使用数据文报无连接地传输数据 - 客户端应用程序

20.7 使用多线程服务器的客户/服务器 Tic-Tac-Toe 游戏

本节要展示我们联网应用的巅峰之作——流行的 Tic-Tac-Toe（即井字棋）游戏，它是用流套接字和客户/服务器技术来实现的。程序包括一个 TicTacToeServer 类（图 20.6），它允许两个 TicTacToeClient（图 20.7）连接到服务器玩游戏。图 20.7 还显示了游戏输出。服务器收到一个客户请求后，就创建 Player 类（图 20.6）的一个对象，以便在单独的执行线程中处理请求。这样，服务器就能处理来自两个客户的请求，每个客户都能独立地玩游戏。服务器为第一个客户分配“X”（代表先手），将“O”分配给第二个客户。游戏过程中，服务器负责维护与棋盘状态有关的信息，使服务器能验证玩家的落棋请求。每个 TicTacToeClient 都维护着用于显示游戏的 Tic-Tac-Toe 棋盘的 GUI 版本。只能在棋盘的一个方格中落棋（显示标记）。

现在讨论服务器端应用程序（图 20.6）。第 168 行实例化一个 TicTacToeServer 对象，并调用它的 execute 方法。TicTacToeServer 构造函数（第 73~91 行）创建数据成员 currentPlayer 和条件变量 turnCondition。服务器用这些成员限制对 validMove 方法的访问，确保只有当前玩家才能落棋。第 82 行创建 gameBeginEvent，它是一个 threading.Event 对象，用于同步游戏的开始。接着，第 84~85 行初始化 Tic-Tac-Toe 棋盘，这是一个含有 9 个元素的列表。注意，棋盘上的每个位置都初始化为 None，表明两个玩家都没有占用任何空格。我们用 0~8 的数字来维护落棋位置（0~2 是第一行，3~5 是第二行，6~8 是第三行）。第 88~90 行准备套接字，以便服务器侦听玩家连接，并显示服务器就绪的消息。

```

1 # Fig. 20.6: fig20_06.py
2 # Class TicTacToeServer maintains a game of Tic-Tac-Toe
3 # for two clients, each managed by a Player thread.
4
5 import socket
6 import threading
7
8 class Player( threading.Thread ):
9     """Thread to manage each Tic-Tac-Toe client individually"""
10
11     def __init__( self, connection, server, number ):
12         """Initialize thread and setup variables"""
13
14         threading.Thread.__init__( self )
15
16         # specify player's mark
17         if number == 0:
18             self.mark = "X"
19         else:
20             self.mark = "O"
21
22         self.connection = connection
23         self.server = server
24         self.number = number
25
26     def otherPlayerMoved( self, location ):
27         """Notify client of opponent's last move"""
28
29         self.connection.send( "Opponent moved." )
30         self.connection.send( str( location ) )
31
32     def run( self ):
33         """Play the game"""

```

```

34
35     # send client message indicating its mark (X or O)
36     self.server.display( "Player %s connected." % self.mark )
37     self.connection.send( self.mark )
38
39     # wait for another player to arrive
40     if self.mark == "X":
41         self.connection.send( "Waiting for another player..." )
42         self.server.gameBeginEvent.wait()
43         self.connection.send(
44             "Other player connected. Your move." )
45     else:
46         self.server.gameBeginEvent.wait() # wait for server
47         self.connection.send( "Waiting for first move..." )
48
49     # play game until over
50     while not self.server.gameOver():
51
52         # get more location from client
53         location = self.connection.recv( 1 )
54
55         if not location:
56             break
57
58         # check for valid move
59         if self.server.validMove( int( location ), self.number ):
60             self.server.display( "loc: " + location )
61             self.connection.send( "Valid move." )
62         else:
63             self.connection.send( "Invalid move, try again." )
64
65         # close connection to client
66         self.connection.close()
67         self.server.display( "Game over." )
68         self.server.display( "Connection closed." )
69
70     class TicTacToeServer:
71         """Server that maintains a game of Tic-Tac-Toe for two clients"""
72
73         def __init__( self ):
74             """Initialize variables and setup server"""
75
76             HOST = ""
77             PORT = 5000
78
79             self.board = []
80             self.currentPlayer = 0
81             self.turnCondition = threading.Condition()
82             self.gameBeginEvent = threading.Event()
83
84             for i in range( 9 ):
85                 self.board.append( None )
86
87             # setup server socket
88             self.server = socket.socket( socket.AF_INET,
89                 socket.SOCK_STREAM )
90             self.server.bind( ( HOST, PORT ) )
91             self.display( "Server awaiting connections..." )
92
93         def execute( self ):
94             """Play the game--create and start both Player threads"""
95
96             self.players = []
97
98             # wait for and accept two client connections
99             for i in range( 2 ):
100                 self.server.listen( 2 )
101                 connection, address = self.server.accept()
102
103                 # assign each client to a Player thread
104                 self.players.append( Player( connection, self, i ) )

```

```

105         self.players[ -1 ].start()
106
107     self.server.close()    # no more connections to wait for
108
109     # players are suspended until player 0 connects
110     # resume players now
111     self.gameBeginEvent.set()
112
113     def display( self, message ):
114         """Display a message on the server"""
115
116         print message
117
118     def validMove( self, location, player ):
119         """Determine if a move is valid--if so, make move"""
120
121         # only one move can be made at a time
122         self.turnCondition.acquire()
123
124         # while not current player, must wait for turn
125         while player != self.currentPlayer:
126             self.turnCondition.wait()
127
128         # make move if location is not occupied
129         if not self.isOccupied( location ):
130
131             # set move on board
132             if self.currentPlayer == 0:
133                 self.board[ location ] = "X"
134             else:
135                 self.board[ location ] = "O"
136
137             # change current player
138             self.currentPlayer = ( self.currentPlayer + 1 ) % 2
139             self.players[ self.currentPlayer ].otherPlayerMoved(
140                 location )
141
142             # tell waiting player to continue
143             self.turnCondition.notify()
144             self.turnCondition.release()
145
146             # valid move
147             return 1
148
149         # invalid move
150         else:
151             self.turnCondition.notify()
152             self.turnCondition.release()
153             return 0
154
155     def isOccupied( self, location ):
156         """Determine if a space is occupied"""
157
158         return self.board[ location ] != None    # an empty space is None
159
160     def gameOver( self ):
161         """Determine if the game is over"""
162
163         # place code here testing for a game winner
164         # left as an exercise for the reader
165         return 0
166
167     def main():
168         TicTacToeServer().execute()
169
170     if __name__ == "__main__":
171         main()

```

```

Server awaiting connections...
Player X connected.
Player O connected.
loc: 0
loc: 4
loc: 3
loc: 1
loc: 7
loc: 5
loc: 2
loc: 8
loc: 6

```

图 20.6 Tic-Tac-Toe 的服务器端应用程序

`execute` 方法（第 93~111 行）循环两次，每次都等待来自客户的一个连接请求。服务器收到一个连接请求后，就创建 `Player` 对象（第 8~68 行），以便将连接作为一个独立的线程进行管理。`Player` 构造函数（第 11~24 行）的参数包括一个 `socket` 对象（代表到客户的连接）、`TicTacToeServer` 对象以及用于标记玩家的一个字符（X 或 O）。第 14 行初始化线程。

服务器创建好每个 `Player` 后（第 104 行），会调用 `Player` 的 `start` 方法（第 105 行）。`Player` 的 `run` 方法（第 32~68 行）控制着发送给客户以及从客户那里接收的信息。首先，方法将客户传递落棋下放到棋盘上的字符传递给客户。然后，方法发送一条消息，提醒客户已经成功连接（第 36~37 行）。接着，第 40~44 行阻塞玩家 X，直到游戏可以开始（即玩家 O 加入）。类似地，第 45~47 行阻塞玩家 O，直到服务器开始游戏。两个玩家都加入游戏后，服务器关闭它的套接字（第 107 行），并调用 `Event` 的 `set` 方法来开始游戏（第 111 行）。

之后，每个 `Player` 的 `run` 方法都会执行自己的 `while` 循环（第 50~63 行）。每次重复这个 `while` 循环时，都会收到一个代表目标落棋位置的字符串，并调用 `TicTacToeServer` 的 `validMove` 方法来判断这个落棋请求是否合法。第 61 行和第 63 行向客户发送消息，指明落棋是否有效。游戏会一直进行，直到 `TicTacToeServer` 的 `gameOver` 方法（第 160~165 行）指出游戏已经结束。游戏终止后，第 66~68 行会关闭到客户的连接，并在服务器上显示一条消息。请注意 `gameOver` 方法的逻辑，它应该能够判断游戏是赢、是输还是平局，这留给读者自行练习。

`validMove` 方法（`TicTacToeServer` 中的第 118~153 行）使用条件变量的 `acquire` 和 `release` 方法限制每次只能由一个玩家落棋。这样可避免两个玩家同时修改游戏的状态信息。如果 `Player` 试图验证并非当前玩家（即允许落棋的玩家）的一次落棋，`Player` 就会进入“waiting”状态，直到轮到该玩家落棋。如果另一个标记已经占据了指定的落棋位置，`validMove` 方法就会返回 0（第 153 行）。否则，服务器在玩家本机显示的棋盘上放置一个标记，并更新 `currentPlayer` 变量。然后，服务器调用 `Player` 的 `otherPlayerMoved` 方法（第 26~30 行）向客户发出通知，调用 `notify` 方法使正在等待的 `Player` 能够验证一次落棋；并返回 1 指明落棋有效（第 143~147 行）。

```

1 # Fig. 20.7: fig20_07.py
2 # Client for Tic-Tac-Toe program.
3
4 import socket
5 import threading
6 from Tkinter import *
7 import Pmw
8
9 class TicTacToeClient( Frame, threading.Thread ):
10     """Client that plays a game of Tic-Tac-Toe"""
11
12     def __init__( self ):
13         """Create GUI and play game"""
14
15         threading.Thread.__init__( self )
16
17         # initialize GUI
18         Frame.__init__( self )
19         Pmw.initialise()

```

```

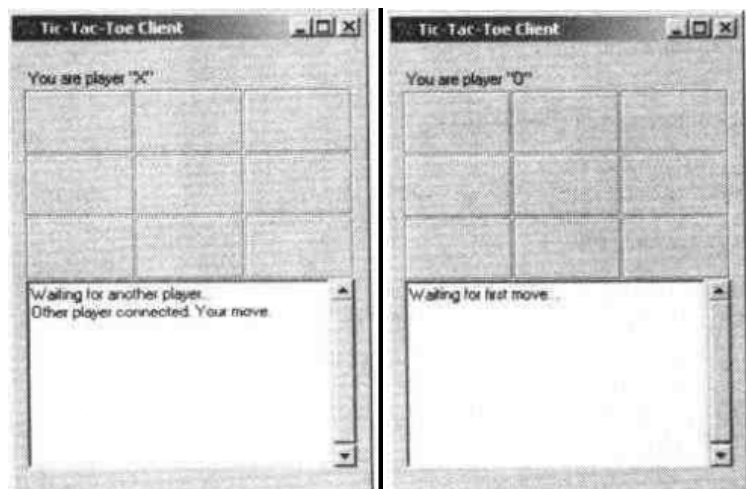
20 self.pack( expand = YES, fill = BOTH )
21 self.master.title( "Tic-Tac-Toe Client" )
22 self.master.geometry( "250x325" )
23
24 self.id = Label( self, anchor = W )
25 self.id.grid( columnspan = 3, sticky = W+E+N+S )
26
27 self.board = []
28
29 # create and add all buttons to the board
30 for i in range( 9 ):
31     newButton = Button( self, font = "Courier 20 bold",
32         height = 1, width = 1, relief = GROOVE,
33         name = str( i ) )
34     newButton.bind( "<Button-1>", self.sendClickedSquare )
35     self.board.append( newButton )
36
37 current = 0
38
39 # display buttons in 3x3 grid beginning with grid's row one
40 for i in range( 1, 4 ):
41     for j in range( 3 ):
42         self.board[ current ].grid( row = i, column = j,
43             sticky = W+E+N+S )
44         current += 1
45
46 # area for server messages
47 self.display = Pmw.ScrolledText( self, text_height = 10,
48     text_width = 35, vscrollmode = "static" )
49 self.display.grid( row = 4, columnspan = 3 )
50
51 self.start() # run thread
52
53 def run( self ):
54     """Control thread to allow continuous updated display"""
55
56     # setup connection to server
57     HOST = "127.0.0.1"
58     PORT = 5000
59     self.connection = socket.socket( socket.AF_INET,
60         socket.SOCK_STREAM )
61     self.connection.connect( ( HOST, PORT ) )
62
63     # first get player 癡 mark ( X or O )
64     self.myMark = self.connection.recv( 1 )
65     self.id.config( text = 'You are player "%s"' % self.myMark )
66
67     self.myTurn = 0
68
69     # receive messages sent to client
70     while 1:
71         message = self.connection.recv( 34 )
72
73         if not message:
74             break
75
76         self.processMessage( message )
77
78     self.connection.close()
79     self.display.insert( END, "Game over.\n" )
80     self.display.insert( END, "Connection closed.\n" )
81     self.display.yview( END )
82
83 def processMessage( self, message ):
84     """Interpret server message to perform necessary actions"""
85
86     # valid move occurred
87     if message == "Valid move.":
88         self.display.insert( END, "Valid move, please wait.\n" )
89         self.display.yview( END )
90
91     # set mark
92     self.board[ self.currentSquare ].config(
93         text = self.myMark, bg = "white" )
94

```

```

95
96     # invalid move occurred
97     elif message == "Invalid move, try again.":
98         self.display.insert( END, message + "\n" )
99         self.display.yview( END )
100         self.myTurn = 1
101
102     # opponent moved
103     elif message == "Opponent moved.":
104
105         # get move location
106         location = int( self.connection.recv( 1 ) )
107
108         # update board
109         if self.myMark == "X":
110             self.board[ location ].config( text = "O",
111                                             bg = "gray" )
112         else:
113             self.board[ location ].config( text = "X",
114                                             bg = "gray" )
115
116         self.display.insert( END, message + " Your turn.\n" )
117         self.display.yview( END )
118         self.myTurn = 1
119
120     # other player's turn
121     elif message == "Other player connected. Your move.":
122         self.display.insert( END, message + "\n" )
123         self.display.yview( END )
124         self.myTurn = 1
125
126     # simply display message
127     else:
128         self.display.insert( END, message + "\n" )
129         self.display.yview( END )
130
131     def sendClickedSquare( self, event ):
132         """Send attempted move to server"""
133
134         if self.myTurn:
135             name = event.widget.winfo_name()
136             self.currentSquare = int( name )
137
138             # send location to server
139             self.connection.send( name )
140             self.myTurn = 0
141
142     def main():
143         TicTacToeClient().mainloop()
144
145     if __name__ == "__main__":
146         main()

```



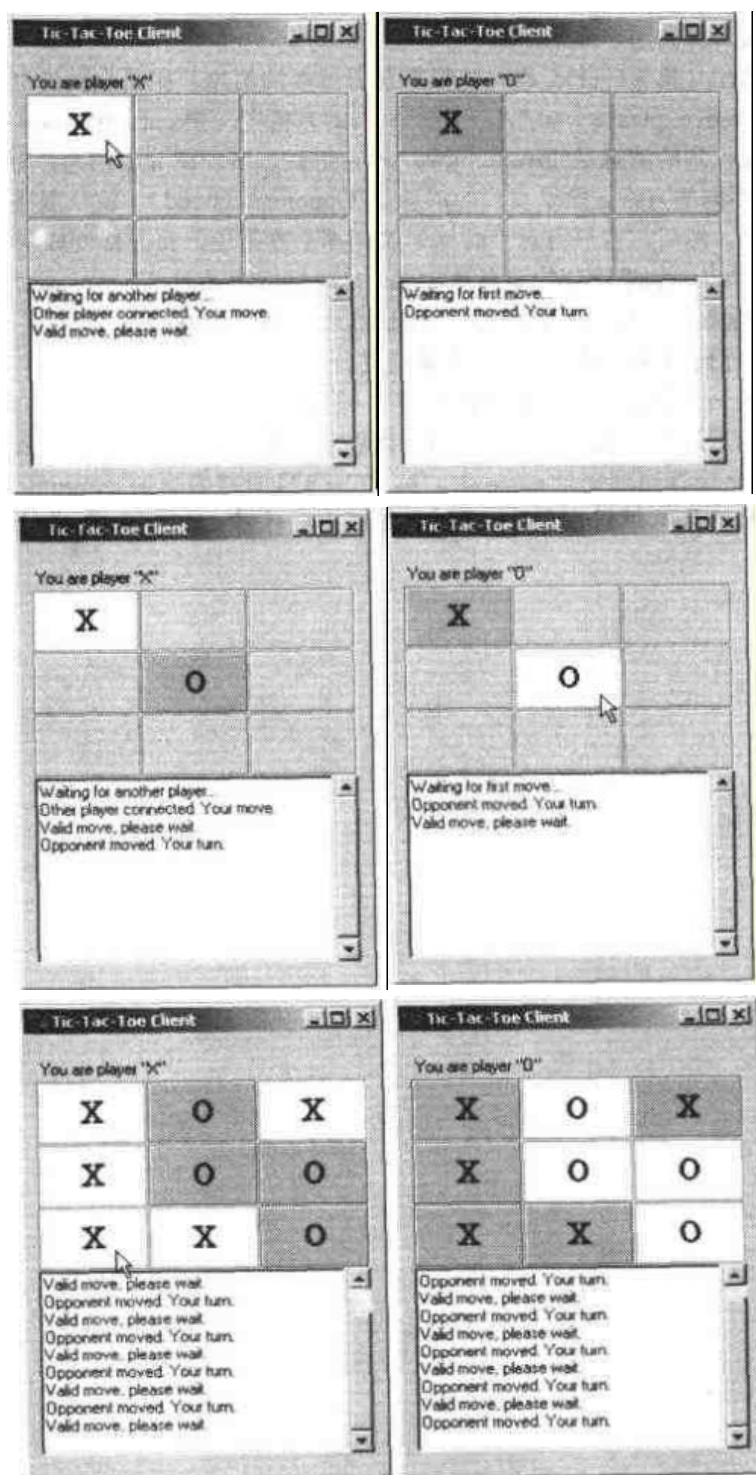


图 20.7 Tic-Tac-Toe 的客户端应用程序

下面接着讨论客户端应用程序。一个 `TicTacToeClient` (图 20.7) 开始执行时, 会创建一个 `Pmw ScrolledText`, 用它显示来自服务器的消息, 并用 9 个 `Tkinter Button` 组件创建棋盘画面。`TicTacToeClient` 类是从 `threading.Thread` 类继承的。通过继承, 一个单独的线程可用于读取服务器发给客户的消息。脚本的 `run` 方法 (第 54~82 行) 打开一个到服务器的连接。建立连接后, 方法从服务器读取 X 或 O 标记字符 (第 65 行), 将 `myTurn` 属性初始化为 0 (第 68 行), 并持续循环以便从服务器读取消息 (第 71~77 行)。消息传给脚本的 `processMessage` 方法进行处理。游戏结束时 (即服务器关闭连接), 第 84~192 行

关闭和每个客户的连接，并向用户显示消息。

`processMessage` 方法（第 84~115 行）解释来自服务器的消息。如果收到的消息是“Valid move.”，客户就显示消息“Valid move, please wait.”，在用户单击的方格中（由 `currentSquare` 指定）放置标记，并将方格变成白色。如果收到的消息是“Invalid move, try again.”，客户就显示消息，并将 `myTurn` 属性设为 1，使用户能单击一个不同的方格。如果收到的消息是“Opponent moved.”，就从服务器接收一个整数，它指明对手将棋落在哪里。然后，客户将对手的标记放到那个方格中，将方格变成灰色，向用户显示一条消息，并将 `myTurn` 设为 1。如果收到的消息是“Other player connected. Your move.”，客户就显示消息，并将 `myTurn` 设为 1。注意，只有在玩家 O 最初连接时，这条消息才会发送给玩家 X（图 20.6，第 42~43 行）。如果收到其他任何消息，客户就显示那条消息。

玩家单击棋盘上的一个方格（一个 Tkinter 按钮），就调用 `sendClickedSquare` 方法（第 117~124 行）。该方法首先检测是否轮到玩家。如果是，第 121 行就获得被单击的那个按钮的名称（调用 Widget 的 `wininfo_name` 方法），并将值存储到变量 `name` 中。第 122~124 行接着更新 `currentSquare` 属性，将落棋信息发送给服务器，并将 `myTurn` 属性设为 0，使玩家不能再进行另一次落棋，除非从服务器那里收到反馈信息。

第 21 章 安全性

学习目标

- 理解基本安全概念
- 理解公钥加密
- 了解流行安全协议，比如 SSL
- 理解数字签名、数字证书、证书颁发机构和公钥基础结构
- 理解 Python 编程的安全问题
- 学会编写受限 Python 代码
- 理解安全系统面临的各种威胁

21.1 概述

电子商务的爆炸性增长，迫使公司和消费者正视 Internet 及网络安全问题。消费者可直接在网上购物、交易股票和处理银行业务。进行这些活动时，需要向网站提交他们的信用卡号、社会安全号以及其他机密信息；商家则通过 Internet 向客户和零售商发送机密信息。与此同时，电子商务正遭受越来越多的安全攻击，公司和客户是最大的受害者。数据窃贼和网上攻击可能破坏文件，甚至使正常商业活动无法开展。所以，成功的电子商务必须有能力防范此类攻击。本章要讨论 Internet 安全性，其中包括对电子事务处理及网络的安全保护，讨论如何进行安全的 Python 编程，探索安全事务处理的基础，以及如何使用最新的技术来保护电子商务。

e-Fact 21.1 国际数据公司（International Data Corporation, IDC）调查显示，1999 年各大公司在安全咨询方面的支出为 62 亿美元，预计到 2003 年底，这个市场的价值将高达 148 亿美元。

现代计算机安全强调了保护电子通信以及保持“网络安全性”的问题及要点。要想保证成功而且安全的事务处理，必须满足 4 项基本要求：保密、完整、身份验证和不可否认。“保密”是指：对于通过 Internet 发送的信息，在没有您的授权时，如何保证不被第三方获取或误传至第三方？“完整”是指：如何保证发送或者接收的信息不会被泄露或修改？“身份验证”是指：一条消息的发送方和接收方如何相互验证对方身份？“不可否认”是指：如何用合法手段证实一条消息已经发出或收到。

除这些要求之外，网络安全性还强调了“可用性”问题：如何保证所连接的网络和计算机系统不间断地工作？

Python 应用程序在运行时，也许能访问本地计算机上的文件。本章解释如何编写安全的、受限环境的 Python 代码。

e-Fact 21.2 根据 Forrester Research 的调查，2002 年和安全有关的花费比 2000 年增加了 55%。

21.2 密码系统古今谈

数据传输渠道天生是不安全的，所以，通过这些渠道传输保密信息时，必须进行某种程度的保护。为保护信息，可对数据进行加密。密码术通过一个密码系统传输数据。“密码系统”是一种对消息进行加密的数学算法（“算法”是一种计算机术语，相当于“操作规程”）。“密钥”是一个由字母或数字构成的字符串，相当于一个密码，并需要输入密码系统。密码系统使用密钥对数据进行处理，确保只有发送者和目标接收者才能理解它。未加密的数据称为“明文”；加密的数据称为“密文”。算法负责数据加密，而密钥相当于一个变量——使用的密钥不同，生成的密文也不同。只有目标接收者才持有相应的密钥，

以便将密文“解密”成明文。

古埃及人最早使用密码系统。当时采用人工方法加密消息，并常常以字母为基础。两种主要的密码系统是置换和移位加密。置换加密中，每出现一个给定的字母就用另外一个不同的字母代替。例如，所有“a”都用“b”代替，所有“b”都用“c”代替，以此类推。按此规则，单词“security”加密成“tfdvsjuz”。最出名的置换密码系统是裘力斯·凯撒发明的，所以今天也把它称为“凯撒密码”。使用凯撒密码，每个字母都被替换成字母表中朝右的第3个字母。例如，“security”被替换成“vhfxulwb”。

在移位加密中，字符顺序被改变。例如在单词“security”中，将从“s”开始每隔一个的所有字母作为密文中的第一个词，剩下的字母则作为第二个词。所以，“security”将加密成“scrt euiz”。复杂加密系统则是这两种加密技术的组合。例如，先进行置换加密，再进行移位加密，“security”就加密成“tdsufvjz”。然而，这些古老加密系统存在的最大问题是：安全性主要依赖于发送者和接收者是否能记这些加密算法，而且是否能保守秘密。这称为“有限算法”，不适合较大的人群。可以想象，如果美国政府通信的安全性建立在每个工作人员的保密之上，加密算法很容易被泄露出来。

现代加密系统是数字化的。它们采用的算法基于一条消息中的单独的“位”或“分组”（分组是位的组合），而非依赖于字母。计算机以“二进制字符串”的形式存储数据，即0和1的一个序列。序列中的每个数位称为一个位或比特。加密和解密密钥是具有指定“密钥长度”的二进制字符串。例如，128位加密系统的密钥长度为128位。更长的密钥可进行更强的加密，要想破解消息，就需要花更多的时间，需要更强的计算能力。

2000年1月之前，美国政府一直都在限制从美国出口的密码系统的强度，也就是限制加密算法所支持的密钥长度。今天，对加密产品的出口限制已被放宽。只要最终用户不是外国政府机构，或者不是明令禁止的国家，任何加密系统都可向其出口。

Python的rotor模块用于加密和解密消息。rotor实际上是一种置换密码系统。图21.1演示了Python中的加密和解密。

```
1 # Fig. 21.01: fig21_01.py
2 # Demonstrating crypto system.
3
4 from Tkinter import *
5 import rotor
6 import string
7
8 class Crypto( Frame ):
9     """Demonstrate the cryptosystem"""
10
11     def __init__( self ):
12         """Create and grid several components into the frame"""
13
14         Frame.__init__( self )
15         self.grid( sticky = W+E+N+S )
16         self.master.title( "Python Encryption and Decryption" )
17         self.master.rowconfigure( 0, weight = 1 )
18         self.master.columnconfigure( 0, weight = 1 )
19
20         self.button1 = Button( self, text = "Encrypt",
21                               width = 15, command = self.encrypt )
22
23         # specify position of Button component button1
24         self.button1.grid( row = 0, column = 1, sticky = W+E+N+S )
25
26         self.button2 = Button( self, text = "Decrypt",
27                               width = 15, command = self.decrypt )
28         self.button2.grid( row = 0, column = 2, sticky = W+E+N+S )
29
30         self.text1 = Text( self, width = 30, height = 15 )
31
32         # text component spans three rows and all available space
33         self.text1.grid( row = 3, column = 1, columnspan = 2,
34                          sticky = W+E+N+S )
35         self.text1.insert( INSERT, "Text" )
```

```

36
37     # makes second row/column expand
38     self.rowconfigure( 1, weight = 1 )
39     self.columnconfigure( 1, weight = 1 )
40
41     self.cipher = rotor.newrotor( "deitelkey", 12 )
42
43     def encrypt( self ):
44         """Encrypt a text"""
45
46         # get text from Text component
47         text = self.text1.get( 1.0, END )
48         text = string.strip( text )
49
50         # encrypt text
51         encryptedText = self.cipher.encrypt( text )
52         self.text1.delete( 1.0, END )
53
54         # display encrypted text
55         self.text1.insert( END, encryptedText )
56
57     def decrypt( self ):
58         """Decrypt a text"""
59
60         # get text from Text component
61         text = self.text1.get( 1.0, END )
62         text = string.strip( text )
63
64         # decrypt text
65         decryptedText = self.cipher.decrypt( text )
66         self.text1.delete( 1.0, END )
67
68         # display decrypted text
69         self.text1.insert( END, decryptedText )
70
71     def main():
72         Crypto().mainloop()
73
74     if __name__ == "__main__":
75         main()

```

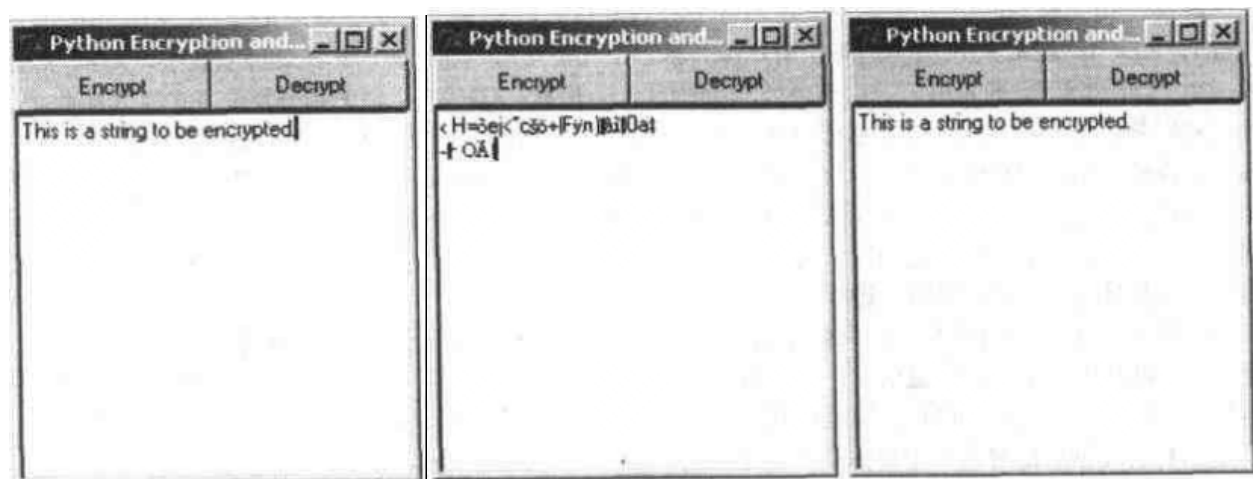


图 21.1 用 Python 的 rotor 模块实现置换密码系统

在构造函数中（第 11~41 行），我们创建一个 GUI，其中含有两个按钮和一个 Text 组件。第 20~28 行在输出窗口顶部放置两个按钮：Encrypt（加密）和 Decrypt（解密）。第 30~35 行在输出窗口添加一个 Text 组件，并将默认文本设为“Text”。第 41 行调用 rotor 模块的 newrotor 函数，从而获得一个 rotor：

```
rotor.newrotor( "deitelkey", 12 )
```

`newrotor` 函数取得两个参数：一个是密钥；另一个是 `rotor` 数。最后返回的是一个 `rotor` 对象。第一个参数（密钥）用于随机生成 `rotor`。第二个参数（`rotor` 数）是可选的。默认情况下，传给 `newrotor` 函数的 `rotor` 数是 6。为了加密一个字符，原始字符被第一个 `rotor` 置换，结果再由第二个 `rotor` 置换。以此类推，直到应用了所有 `rotor`。

用户单击 `Encrypt` 函数，就会调用 `encrypt` 函数（第 43~55 行）。第 47 行从 `Text` 组件获得文本。为获得密文，需要将明文传给 `encrypt` 函数：

```
encryptedText = self.cipher.encrypt( text )
```

第 55 行将加密的文本放到 `Text` 组件中。单击 `Decrypt` 函数，会调用 `decrypt` 函数（第 57~69 行）。第 61 行从 `Text` 组件获得文本。为获得解密的文本，需要将密文传给 `decrypt` 函数：

```
decryptedText = self.cipher.decrypt( text )
```

第 69 行将解密的文本放到 `Text` 组件中。

21.3 加密密钥

过去要想维持安全计算环境，采用的是“对称密码”（或“加密码”）。它采用相同的加密密钥对消息进行加密和解密（图 21.2）。在这种情况下，发送者使用加密密钥对一条消息进行加密，将加密的消息发送给目标接收者。在这个过程中，最基本的问题是通信双方事先必须找到一种安全交换密钥的方法。一种方法是通过信使，或使用像美国联邦快递（FedEx）这样的邮政服务。这种方法在两个人之间通信时是有效的，但在大型网络中通信时，就显得效率低下，而且不算是真正安全的。密钥通过不安全的渠道在发送者和接收者之间传递时，一旦被拦截，就会破坏消息的保密性和完整性。另外，由于双方都用相同的密钥对消息加密和解密，所以很难证实具体是哪一方创建的消息。最后，为了保证与每个接收者都进行保密通信，需要为每个接收者都使用不同的加密密钥。所以，许多大型组织不得不维护大量加密密钥。

解决密钥交换问题的另一种方法是创建中心颁发机构，称为“密钥分发中心”（Key Distribution Center, KDC）。KDC 与网络中的每个用户都共享一个（不同的）加密密钥。在这种系统中，KDC 要生成一个“会话密钥”，把它用于一次事务处理（图 21.3）。接着，KDC 将会话密钥分发给发送者和接收者，并用它们分别与 KDC 共享的加密密钥进行加密。例如，假定商户和客户想进行一次安全的交易，它们各自拥有独一无二的加密密钥，并与 KDC 共享。KDC 为此次交易生成一个会话密钥，向商户发送会话密钥，并用商户之前与中心共享的加密密钥进行加密。然后，将同一个会话密钥发送给客户，并用客户之前与中心共享的加密密钥进行加密。商户和客户都拿到此次交易的会话密钥后，相互之间就可安全地通信，使用共享的会话密钥对消息进行加密。

使用密钥分发中心，减少了在网络中每个用户之间来回传递加密密钥的数量（通过邮件或快递服务）。此外，与网络中的每个用户通信时，每次都可使用一个新的加密密钥。这显著提高了网络总体安全性。但是，一旦密钥分发中心自身的安全被破坏，整个网络就不再是安全的。

一种普遍使用的对称加密算法是“数据加密标准”（Data Encryption Standard, DES）。IBM 的 Horst Feistel 发明了 Lucifer 算法。20 世纪 70 年代，美国政府和美国国家安全局（NSA）选中了它，并将其命名为 DES。DES 密钥长度为 56 位，采用 64 位分组对数据进行加密。这种类型的加密称为“分组密码系统”（Block Cipher）。这种系统要依据一条原始消息创建位组（比特组），将其当作一个整体，再向其应用加密算法。换言之，不是对单独的位应用算法。这样便减少了所需的计算机处理能力和时间，同时保持令人满意的安全级别。多年来，DES 一直是美国政府和“美国国家标准协会”（ANSI）设定的加密标准。然而，随着技术及计算能力的飞速发展，DES 也不再安全。20 世纪 90 年代末，专业化的“DES 破解机”研制成功，几小时即可成功破解 DES 密钥。所以，老的对称加密标准被替换成“Triple DES”

(3DES)，它是 DES 的一个变种，实际上是并排的 3 个 DES 系统，每个系统都有自己的加密密钥。虽然 3DES 提高了安全性，但由于要经历 3 遍 DES 算法，导致系统性能下降。因此，美国政府后来又换用一种新的、更安全的对称加密标准，名为“高级加密标准”(Advanced Encryption Standard, AES)。负责为美国政府建立加密标准的“美国国家标准和技术协会”(NIST) 经过考核，将 Rijndael 选定为 AES 的加密方法。Rijndael 是一种“分组密码系统”，发明人是比利时的两位博士 Joan Daemen 和 Vincent Rijmen。Rijndael 支持的密钥长度和分组大小是 128、192 或 256 位。在 AES 的总共 5 种候选加密方法，Rijndael 之所以胜出，是因为它具有高度安全性、出色的性能、极高的运算效率、强大灵活性以及对计算系统的内存的较低要求。要想了解 AES 的详情，请访问 csrc.nist.gov/encryption/aes。

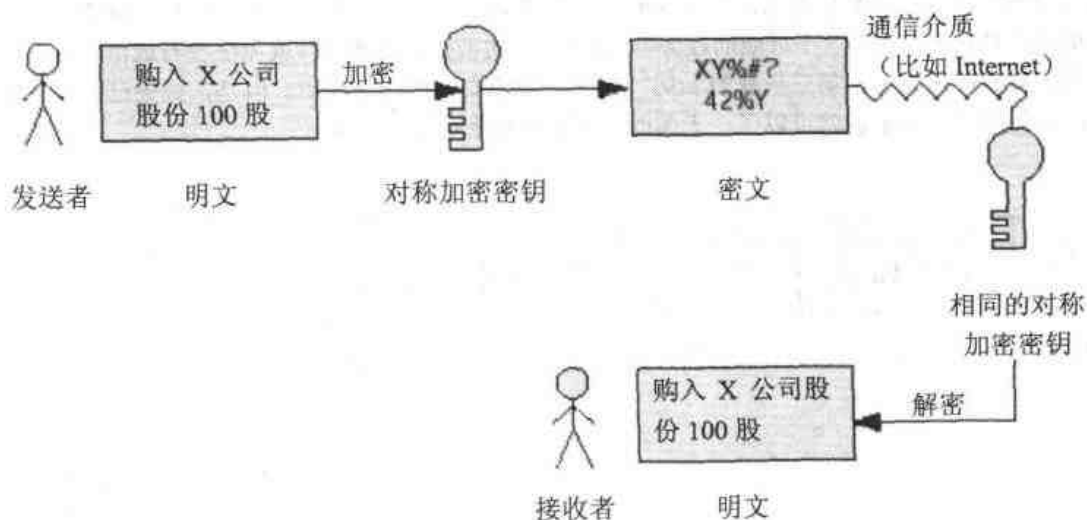


图 21.2 用加密密钥对消息进行加密和解密

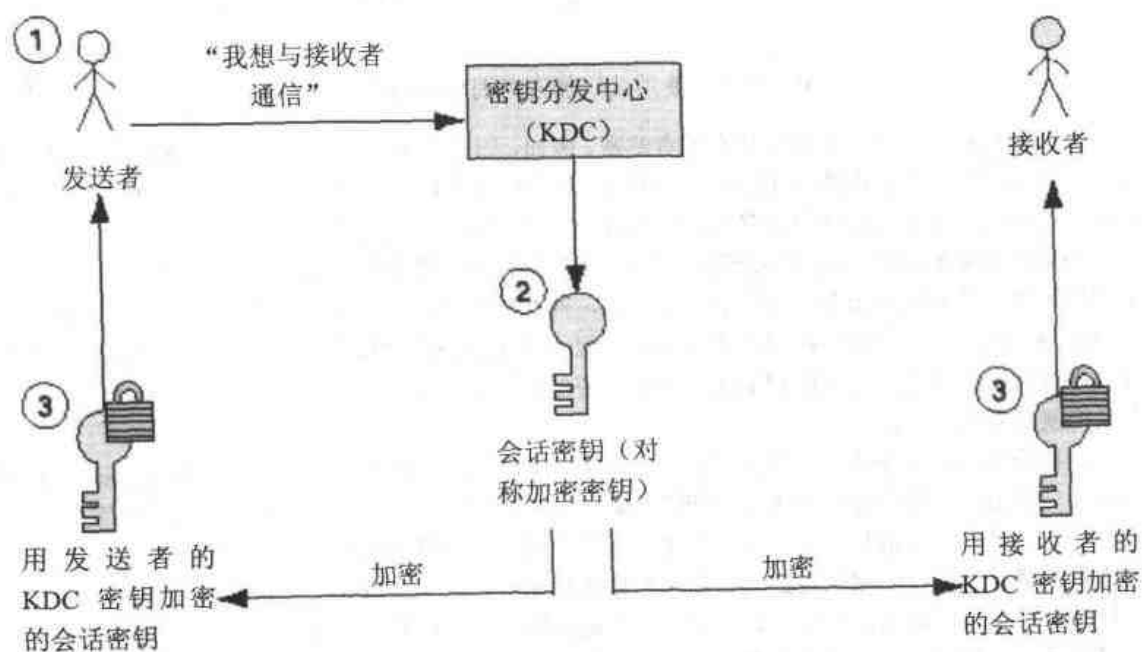


图 21.3 通过密钥分发中心分发一个会话密钥

21.4 公钥加密

1976 年，斯坦福大学的研究员 Whitfield Diffie 和 Martin Hellman 共同开发了能够解决密钥安全交换问题的公钥加密技术。公钥加密是非对称的。它使用两个反向关联的密钥：公钥和私钥。私钥由用户个人秘密保存，公钥则自由公开。用公钥对消息加密，只有相应的私钥才能解密，反之亦然（图 21.4）。通信双方都有自己的公钥和私钥。为安全发送消息，发送方使用接收方的公钥对消息加密；收到消息后，接收方用自己的私钥对消息解密。因为只有接收者知道自己的私钥，所以其他人无法识别这条消息。这样便实现了消息的保密性。对于安全的“公钥算法”，它最突出的特点就是：不能从公钥推导出私钥。尽管这两种密钥在数学上是相关的，但要从其中一个推导出另一个，需花费的时间和计算资源是可观的，要想推导出私钥，简直是“不可能的任务”。没有正确的密钥，外部实体就无法参与通信。整个过程的安全性基于对私钥的保密。第三者获得私钥，整个系统的安全性就会崩溃。但是，如果已知系统完整性被破坏，只需改变一下密钥就可以了，无需改变完整的加密或解密算法。

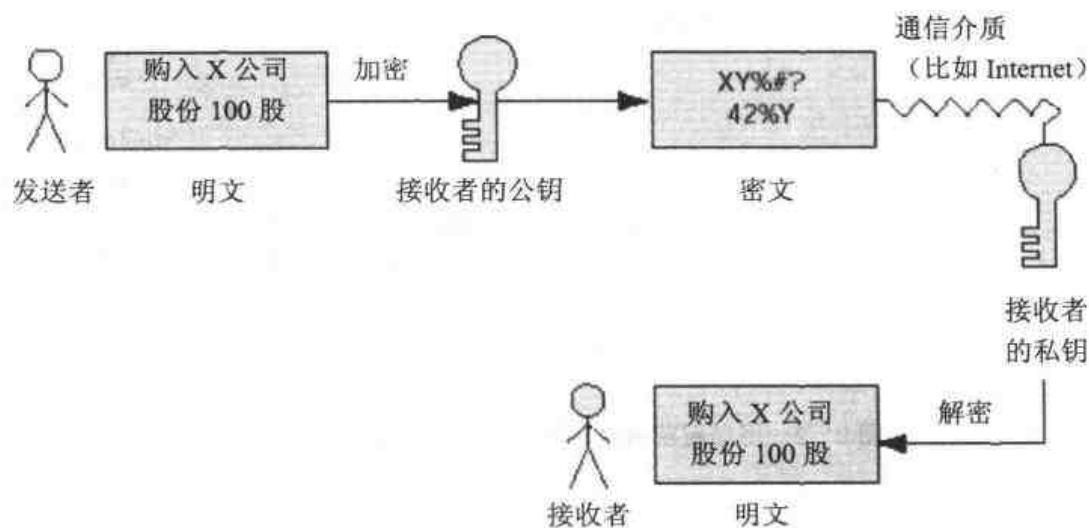


图 21.4 使用公钥加密对消息进行加密和解密

公钥和私钥都用来对消息加密或者解密。例如，如果一个客户使用某个商户的公钥加密一条消息，只有商户才能用他自己的密钥来解密它。所以，商家的身份得到了验证，因为只有商户自己才知道私钥。不过，商户没有办法验证客户的身份，因为客户使用的加密密钥是公开的。

如果用来解密的是发送者的公钥，用来加密的是发送者的私钥，就可验证发送者的身份。例如，假定客户用自己的私钥加密发给商户的一条消息，商户再用客户的公钥对其进行解密，那么商户就能验证客户身份。然而，虽然能证明发送者的身份，但不能保证消息的机密性，任何人都可使用发送者的公钥对其进行解密。不过，只要商户能保证用于解密的公钥是属于客户的，而不是属于一个冒充的客户，这个系统就是有效的。

两种公钥加密方法实际可以共用，目的是验证通信双方的身份（图 21.5）。假定商家希望向客户安全发送一条消息，并保证只有该客户能够读取消息；同时，假定商家想向客户证明确实是自己（而不是第三者）发送的这条消息，那么商家首先要用客户的公钥对消息加密。这个步骤可保证只有那个客户才能读取消息。然后，商家用自己的私钥对结果进行加密。这个步骤是为了证实商家的身份。客户则按相反顺序对消息进行解密。首先，客户使用商家的公钥对消息解密。由于只有商家才能使用反向关联的私钥对消息进行加密，所以这个步骤能验证商家身份。然后，客户用自己的私钥对刚才获得的结果进行下一级解密。这个步骤确保消息传输过程中的机密性，因为只有客户才拥有对消息进行解密的密钥。尽管

这个系统实现了高度安全的事务处理，但由于设置费时费力，所以阻碍了它的广泛应用。

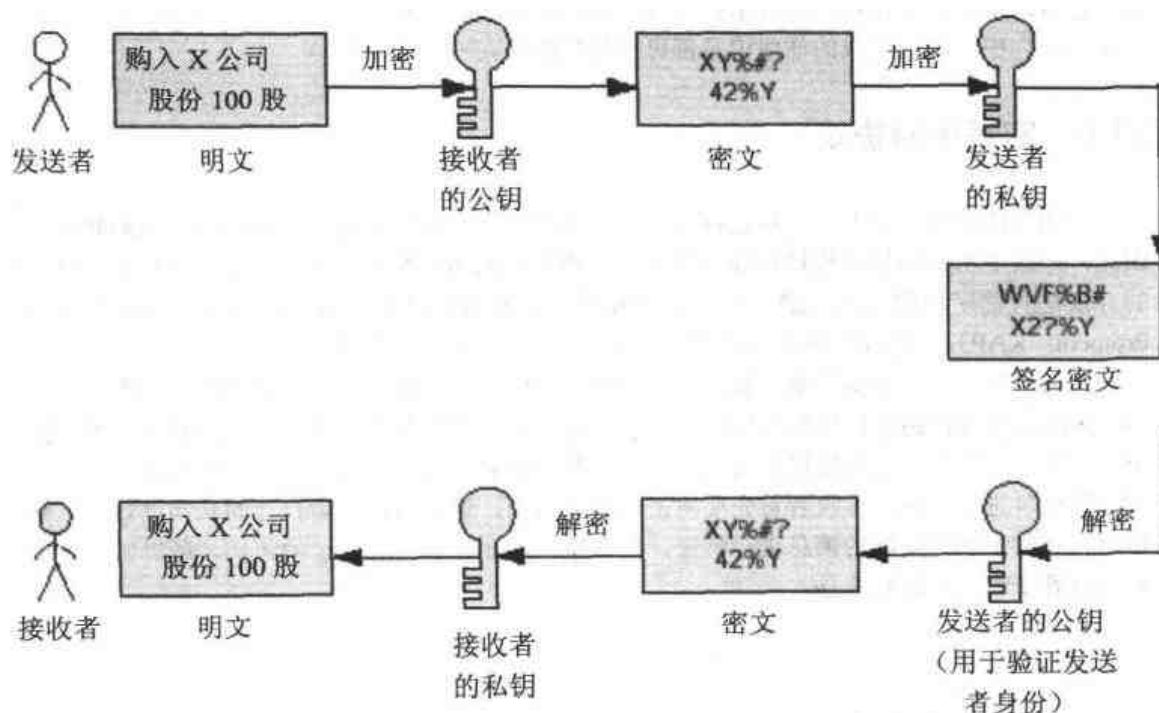


图 21.5 用公钥算法进行身份验证

最常用的公钥算法是 RSA, 1977 年由麻省理工学院 (MIT) 3 名教授 Ron Rivest, Adi Shamir 和 Leonard Adleman 等三名教授研制成功。今天, 全球许多传统公司以及领先的电子商务公司都在使用这个加密和身份验证技术。随着 Internet 和万维网的普及, 他们当年进行的研究显得越来越重要, 而且在电子商务中发挥着关键作用。许多流行的 Internet 应用程序都集成了他们的加密产品, 其中包括 Web 浏览器、电子商务服务器以及电子邮件系统等等。网上进行的最安全的电子商务事务处理以及通信采用的都是 RSA 产品。要想了解 RSA、加密和安全性的详情, 请访问 www.rsasecurity.com。

PGP (Pretty Good Privacy) 是另一种公钥加密系统, 用于加密电子邮件和文件。PGP 设计于 1991 年, 设计者是 Phillip Zimmermann。PGP 也可提供数字签名 (参见 21.8 节), 从而确定电子邮件或其他公开作品的作者身份。PGP 建立在一个“可信网络”的基础上。网络中每个客户都可担保另一个客户的身份, 以证实公钥的所有者。这种可信网络可验证每个客户的身份。如用户知道一个公钥持有者的身份 (通过私人联系或其他安全方式), 就能用自己的密钥去签署它, 从而对密钥进行证实。随着越来越多的用户证实其他人的密钥, 可信网络会变得越来越大。要想了解 PGP 的更多信息以及下载软件的免费拷贝, 请访问 PGP 的“MIT 分发中心”, 网址是 web.mit.edu/network/pgp.html。

21.5 密码破解

即便密钥是保密的, 系统安全性仍有可能被破坏。在不知道解密密钥的前提下, 试图对密文进行解密的过程称为“密码破解”。商业化的加密系统经常要邀请密码破解专家进行研究, 确保系统能抵抗“密码破解攻击”。最常见的攻击形式就是分析加密算法, 找出加密密钥位与密文位的联系。通常, 这些联系建立在统计学基础上, 而且集成了和明文本身没有任何关系的知识。攻击的最终目标就是找出这些联系, 并根据密文推导出密钥。

掌握足够多的密文, 密文与密钥之间的弱统计学趋势就会逐渐暴露, 让破解者更可能推导出密钥。

因此，正确的密钥管理和过期设置有助于防止这种攻击。如果一个密钥长期不予更换，就会通过它生成越来越多的密文，这使攻击者能掌握充分的资料完成破解。只要攻击者破解出一个密钥，在那个密钥的“生命期”中，用它生成的任何消息都可被轻松解密。

21.6 密钥协商协议

公钥算法的缺点是发送大量数据时效率不够高。它需要耗用大量计算能力，从而妨碍了通信速度。因此，一般不用公钥算法代替对称密钥算法。相反，公钥算法允许双方约定一个密钥，以便通过不安全的介质进行加密通信。双方通过不安全的介质交换密钥的过程称为“密钥协商协议”（Key Agreement Protocol, KAP）。“协议”用于订立通信规则，即确定具体使用的加密算法。

最常用的密钥协商协议是“数字信封”（图 21.6）。使用数字信封，消息用一个密钥进行加密（步骤 1），再用公钥对密钥进行加密（步骤 2）。发送者将加密的密钥附加到加密的消息上，再向接收者发送整个包。另外，发送者可在发送前对这个包进行数字签名，向接收者证实自己的身份（参见 21.8 节）。要想对这个包进行解密，接收者首先要将密钥解密出来，这是用接收者自己的私钥进行的。然后，接收者使用得到的密钥对实际的消息进行解密。只有接收者才能解出加密过的密钥，所以发送者可保证只有目标接收者才能正常读取消息。

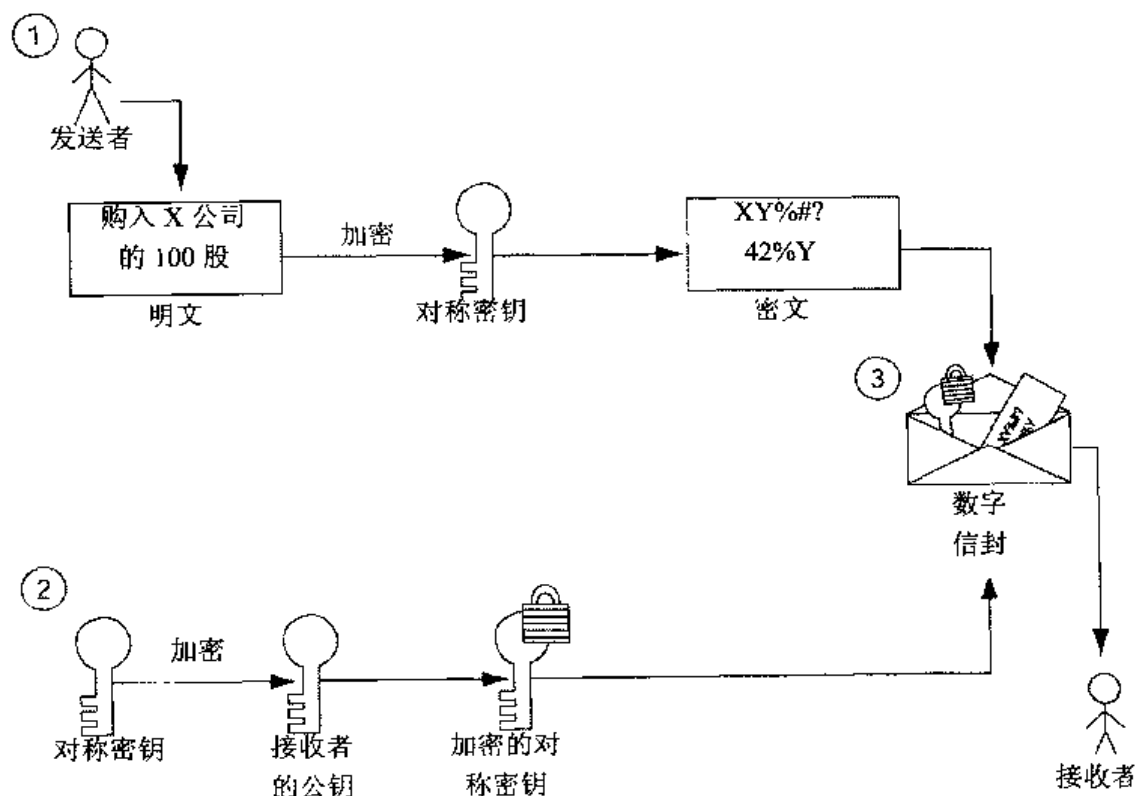


图 21.6 创建数字信封

21.7 密钥管理

要保持加密系统的安全，私钥一定要保密。如果密钥管理机制不佳，例如因为管理随意而导致密钥被盗，会为系统带来严重的安全威胁。

进行密钥管理时，最重要的工作就是“密钥生成”——即创建密钥的过程。恶意的第三者可能使用

所有可能的解密密钥来尝试破解一条消息，这称为“野蛮破解”(Brute-force Cracking)。一些密钥生成算法由于设计不佳，造成可用的密钥组合数量过少。而假如子集太小，野蛮攻击就很容易成功。所以，密钥生成程序必须确保生成大量的密钥组合，而且尽可能做到随机。密钥长度则要保证攻击者不可能在短时间内尝试所有组合。

21.8 数字签名

数字签名和手写签名一样，都是为了证实签名者的身份。在公钥加密系统中，人们利用数字签名来解决身份验证和数据完整性这两个问题。数字签名可验证发送者的身份，而且就像手写签名那样，它是很难模仿的。

要创建数字签名，发送者首先获得原始的明文消息，把它传给一个哈希函数，从而通过数学计算为消息赋予一个哈希值。“单向哈希函数”可生成一个字符串，它是输入文件所特有的。“安全哈希算法”(SHA-1)是目前采用的哈希函数标准。使用 SHA-1，明文消息“Buy 100 shares of company X”(购入 X 公司股份 100 股)所生成的哈希值是：D8 A9 B6 9F 72 65 0B D5 6D 0C 47 00 95 0D FD 31 96 0A FD B5。MD5 是另一种流行的哈希函数，发明者是 Ronald Rivest，它通过输入文件的一个 128 位的哈希值来验证数据完整性。图 21.7 的交互式会话演示了 SHA-1。

```
Python 2.2b2 (#26, Nov 16 2001, 11:44:11) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sha
>>> m1 = sha.new( "Buy 100 shares of company X" )
>>> print m1.hexdigest()
d8a9b69f72650bd56d0c4700950dfd31960afdb5
>>> m2 = sha.new()
>>> m2.update( "Buy 100 shares " )
>>> print m2.hexdigest()
4269dec883d1a763250d701c881defd34171809a
>>> m2.update( "of company X" )
>>> print m2.hexdigest()
d8a9b69f72650bd56d0c4700950dfd31960afdb5
>>>
```

图 21.7 在 Python 中进行 SHA-1 哈希处理

home.istar.ca/~neutron/messagedigest 提供了 SHA-1 和 MD5 的例子。在这个站点，用户可在程序中输入文本或文件以生成哈希值，即所谓“消息摘要”(Message Digest)。如果多个消息具有相同的哈希值，就会产生“冲突”。不过，在 SHA-1 和 MD5 中产生冲突的概率是微乎其微的，近似于不可能。一般也不可能根据哈希值推导出原始消息，或者出现两条具有相同哈希值的消息。

随后，发送者使用自己的私钥对消息摘要进行加密。这个步骤会创建一个数字签名，并验证发送者身份，因为只有私钥的所有者才可加密消息。包含数字签名、哈希函数以及原始消息（用接收者的公钥进行加密）的一条消息将发送给接收者。接收者使用发送者的公钥解出原始数字签名，从而得到消息摘要。然后，接收者用自己的私钥解出原始消息。最后，接收者将哈希函数应用于原始消息。如果原始消息的哈希值与签名中包括的消息摘要相符，就表明消息是完整的，在传输过程中没有被改动过。

数字签名和手写签名有一个根本区别：手写签名独立于被签名的文档。所以，如果有人伪造手写签名，就可用它伪造多个文档。数字签名则根据文档内容生成，所以签署的每个文档都具有不同数字签名。

数字签名不能证明消息确实已经发送。考虑以下情况：承包商向公司发送了一份含有数字签名的合同，后来又想取消该合同。为此，承包商可以故意泄露自己的私钥，然后抵赖说这个数字签名来源于私钥窃贼。要想实现不可抵赖的数字签名，可采用“时间标记”(Timestamping)，也就是为数字化文档绑定时间/日期。例如，假定公司和承包商就一份合同进行协商，公司要求承包商用数字签名签署该合同，再由一个称为“时间标记代理”的第三方机构对文档进行数字化的时间标记。承包商将含有数字签名的文档发送给时间标记代理机构。注意，文档本身的内容仍是保密的，因为时间标记代理机构看到的只是

加密的、经过数字签名的文档（而不是原始明文文档）。代理机构将接收时间和日期附加到加密的、数字签名的文档，并用自己的私钥对整个“包”进行数字签名。时间标记只能由代理机构改动，其他任何人都无法代劳，因为他们没有代理机构的私钥。除非承包商在文档做上时间标记之前报告私钥被盗，否则就无法抵赖自己签署的文档。发送者也可要求接收者对文档进行数字签名，并做上时间标记，以证明确实收到了这个文档。要想了解时间标记的详情，请访问 AuthentiDate.com。

美国政府规定的数字化身份验证标准称为“数字签名算法”（DSA）。美国政府已通过一项法律，承认数字签名和手写签名具有相同的法律效力。这项立法将极大地促进电子商务。请访问 www.itaa.org/infosec，更多地了解美国政府和信息安全有关的立法。其他信息还可访问：

[thomas.loc.gov/cgi-bin/bdquery/z?d106:hr.01714:](http://thomas.loc.gov/cgi-bin/bdquery/z?d106:hr.01714)
[thomas.loc.gov/cgi-bin/bdquery/z?d106:s.00761:](http://thomas.loc.gov/cgi-bin/bdquery/z?d106:s.00761)

21.9 公钥基础结构

公钥加密的问题在于，持有一套密钥的任何人都可能盲目地认为另一个人的身份是真实的。例如，假定客户向网上的某个商家发送一份订单，那么怎样才能知道一个网站确实属于那个商家，而不是其他人向盗窃信用卡信息而冒充该商家而建立的网站？“公钥基础结构”（Public Key Infrastructure, PKI）提供了解决这个问题的一种方案。PKI 将公钥加密与“数字证书”和“证书颁发机构”集成到一起。

e-Fact 21.3 Aberdeen Group 预测，到 2003 年底，“全球 2000 强”中约有 98% 的公司会实现 PKI 方案。

数字证书是用于标识用户的一份数字文档，由证书颁发机构（CA）颁发。数字证书包括主体（要认证的公司或个人）的名称、主体的公钥、一个序列号、一个截止日期、受信任的证书颁发机构的签名以及其他任何相关信息，如图 21.8 所示。CA 可以是金融机构或者其他受信任的第三方实体，比如 VeriSign。一经颁发，数字证书就可公开，并在 CA 的证书数据库中存档。

CA 在签署证书时，需要使用 CA 自己的私钥来加密主体的公钥或者公钥的一个哈希值。CA 必须验证每个主体的公钥。所以，用户必须信任 CA 的公钥。通常，每个 CA 都是一个证书颁发机构层次结构的一部分。该层次结构类似于一个信任链，每个环节都依赖另一个环节提供身份验证信息。CA 层次结构正是由许多 CA 构成的一条链子，最开始是根证书颁发机构，它是 Internet 策略注册机构（Internet Policy Registration Authority, IPRA）。IPRA 使用根钥签署证书。根钥只为策略创建机构签署证书。后者是建立数字证书获取策略的机构。策略创建机构依次为 CA 签署数字证书。CA 再为个人和其他组织签署数字证书。CA 负责身份验证，所以它在颁发一份数字证书之前，必须仔细地检查信息。VeriSign 曾由于人为错误而向冒名顶替的微软员工颁发了两份数字证书。这种错误会造成严重危害：颁发不当的证书会导致用户不知不觉将恶意代码下载到自己的机器（请参见后文补充内容“身份验证：Microsoft Authenticode”）。

VeriSign 是一家全球领先的证书颁发机构。要想了解有关 VeriSign 的详情，请访问 www.verisign.com。

e-Fact 21.4 根据 Identrus 的报告，要建设一个数字证书基础结构，金融机构需要花一年时间和 500~1000 万美元。Identrus 是一个全球金融公司联盟，为受信任的 B2B 电子商务提供了一个基本框架。



图 21.8 VeriSign 数字证书

身份验证: Microsoft Authenticode

如何知道网上订购的软件是安全而且没有被改动? 如何保证下载的不是一个会把计算机所有内容都清除的病毒? 您信任软件的来源吗? 随着电子商务的蓬勃发展, 软件公司开始在线提供产品, 客户可将软件直接下载到计算机。人们用安全技术保证下载的软件是受信任的。Microsoft Authenticode 和 VeriSign 数字证书(或者“数字标识”)能验证软件出版商的身份, 并检测软件是否被改动。Authenticode 是嵌入 Microsoft Internet Explorer 中的一种安全特性。

为使用 Microsoft Authenticode 技术, 每个软件出版商都必须申请专门用于出版软件的数字证书; 可通过 VeriSign (参见 21.9 节) 这样的证书颁发机构获得证书。申请证书时, 软件出版商必须提供自己的公钥和标识信息, 并签署一份不得发行有害软件的协议。因为下载认证出版商所提供软件而造成损害的用户就有了法律依据。

Microsoft Authenticode 使用数字签名技术签署软件 (参见 21.8 节)。签署的软件和出版商的数字证书证明软件是安全的, 而且没有被改动过。

用户下载文件时, 屏幕上会出现一个对话框, 显示数字证书和证书颁发机构的名称, 同时还提供了一些链接, 可访问出版商和证书颁发机构, 便于用户在确定下载软件之前, 了解有关各方的详情。如果 Microsoft Authenticode 判断出软件被破坏, 会终止事务处理。

要想知道 Microsoft Authenticode 的详情, 请访问以下网站:

msdn.microsoft.com/workshop/security/authcode/signfaq.asp
msdn.microsoft.com/workshop/security/authcode/authwp.asp

周期性改变密钥对有助于维护系统的安全性, 因为私钥可能在不知不觉中被盗取。密钥对使用时间越长, 就越容易被攻击和破解。因此, 为了督促用户改变密钥对, 数字证书在创建时要指定一个截止日期。如果在这个日期之前私钥泄密, 可以取消数字证书, 用户将得到一个新的密钥对和一份新数字证书。取消和吊销的证书放在“证书吊销列表”(Certificate Revocation List, CRL)中。CRL 由证书颁发机构统一保存。用户一旦怀疑自己的私钥被盗, 必须马上报告, 这是因为“不可抵赖”问题导致证书所有者要为数字证书所牵涉的一切事情负责。美国有些州已经承认数字签名的合法性, 所以证书已经合法地将证书所有者与证书牵涉到的任何事务联系起来。

CRL 类似传统纸质信用卡号码吊销单, 这种列表经常出现在超市的 POS 机器旁边。通过它可方便地验证证书的有效性。CRL 的一种替代品是“在线证书状态协议 (Online Certificate Status Protocol, OCSP), 它能实时验证一份证书。OCSP 技术目前仍在开发阶段。“X.509 Internet Public-Key Infrastructure Online Certificate Status Protocol—OCSP”一文概述了 OCSP, 网址是 <ftp.isi.edu/in-notes/rfc2560.txt>。

仍有许多人认为电子商务是不安全的。事实上, 用 PKI 和数字证书进行事务处理时, 比通过电话线、邮件或者用信用卡付款等私有信息交换方式都要安全。在饭馆就餐, 让侍者拿您的信用卡到后台结账, 怎么知道他没有记下您的信用卡信息? 相比之下, 大多数在线事务处理的密钥算法都几乎是不可能破解的。有人对公钥加密的密钥算法做过估计, 结论是数百万台并行工作的计算机都不可能在 100 年内破译。当然, 随着计算能力的迅速提高, 今天最强的密钥算法也可能在未来被破解。

数字证书已嵌入许多电子邮件软件。例如在 Microsoft Outlook 中, 可选择【工具】菜单, 再选择【选项】, 单击【安全】选项卡。在对话框底部, 可看到【获取数字标识】选项。选择它将连接一个微软网站, 上面列出了几个全球性证书颁发机构的链接。拥有一份数字证书后, 就可为电子邮件加上数字签名了。

要获得用于个人电子邮件的数字证书, 请访问 www.verisign.com 或者 www.thawte.com 网站。VeriSign 提供 60 天免费试用, 也可采取缴纳年费的方式购买服务。Thawte 公司为个人电子邮件提供免费数字证书。虽然也可向 VeriSign 和 Thawte 公司购买 Web 服务器专用证书, 但价格比电子邮件证书贵得多。

智能卡

PKI 成长速度最快的一项应用是智能卡。它的大小如同一张信用卡, 具有多种用途, 从身份验证一直到数据存储。最流行的智能卡是内存卡和微处理器卡。内存卡的作用就像软盘, 微处理器卡则类似于微型计算机, 配备操作系统、安全机制和存储功能。智能卡还有多种接口格式, 通过它们与读卡设备交互。其中一种是接触式接口, 智能卡将插入读卡设备, 设备和卡之间需要建立物理接触。另一种是非接触式接口, 通过卡上集成的无线设备将数据传输到读卡设备, 设备和卡之间无需物理接触。

智能卡存储着私钥、数字证书和实现 PKI 所需的其他信息。另外还存有信用卡号、个人联系信息等。每张智能卡都和个人身份识别号 (PIN) 匹配。这样便提供了双重保护, 要想访问卡上存储的信息, 除必须持有卡之外, 还必须知道正确的 PIN。作为另一级保护, 一些微处理器卡可在发现恶意攻击试图时, 主动删除或破坏存储的数据。智能卡易于携带, 而且同一张卡可访问多个设备的信息。

21.10 安全协议

通过 Web 从事电子商务的每个人都需要关注个人信息的安全性。本节将讨论网络安全协议, 比如网际安全协议 (IPSec); 以及传输层安全协议, 比如安全套接字层 (SSL)。网络安全协议保护网络之间的通信; 传送层安全协议则用于建立安全连接以便传输数据。

21.10.1 安全套接字层 (SSL)

虽然 SSL 不是专为保护事务处理而设计的, 但目前大多数电子商务都用 SSL 进行安全的网上事务处理。SSL 的设计宗旨是保护万维网连接。SSL 协议由 Netscape Communications 公司开发, 是一种非赢利性协议, 通常用于保护 Internet 和 Web 上两台计算机之间的通信。SSL 内建于许多 Web 浏览器 (包括 Netscape Communicator 和 Microsoft Internet Explorer) 和其他软件产品中。它工作于 Internet 的 TCP/IP 通信协议和应用程序软件之间。

Internet 标准响应中, 发送者的消息传给一个套接字, 套接字通过网络收发信息。TCP/IP (传输控制协议/网络互联协议) 是一组协议的统称, 它将计算机和网络连接到 Internet 这个全球最大的“网间网”。大多数 Internet 数据传输以“数据包”的形式进行。在发送端, 一条消息的数据包按顺序编号, 并为每

个数据包附加错误控制信息。IP 主要负责路由传递数据包，避免通信阻塞，所以每个包都可能在 Internet 上以一个不同的路由传递。数据包的目的地由 IP 地址决定，IP 地址是一种号码，用于在网络上标识一台计算机，类似于家庭地址。在接收端，TCP 确保所有数据包都已抵达，将它们还原为正确的顺序，并判断数据包是否被破坏。如果数据包被不经意地改动，或任何数据丢失，TCP 会请求重发。但是，TCP 不能判断数据包是否在传输过程中被恶意改动；恶意的数据包可能伪装成有效的数据包。所有数据都成功抵达 TCP/IP 后，消息会传给接收端的套接字。套接字将消息还原成应用程序可识别和使用的格式。利用 SSL 进行事务处理时，套接字则由公钥密码系统进行保护。

SSL 利用 RSA 算法和数字证书实现了公钥技术，以验证服务器在事务处理中的身份，并对通过 Internet 从一方传输到另一方的保密信息提供保护。SSL 事务处理无需客户身份验证；许多服务器只需一个有效信用卡号码即可处理购物请求。首先，客户要向服务器发送一条消息。服务器对此进行响应，并把它的数字证书发送给客户，以验明服务器身份。使用公钥密码系统进行安全通信时，客户和服务器要协商好事务处理中所用的会话密钥。这个会话密钥只在本次事务处理中使用。一旦设好密钥、会话密钥和数字证书，就可继续在客户和服务器之间通信。加密的数据通过 TCP/IP 传输，这与普通数据包通过 Internet 传输没有区别。然而，用 TCP/IP 发送一条消息之前，SSL 协议会将信息分解成“分组”，对其进行压缩和加密。数据通过 TCP/IP 抵达接收者之后，SSL 协议要对数据包进行解密，并解压缩和重新组装数据。这些额外的处理在 TCP/IP 和应用程序之间提供了一个额外的安全层。SSL 主要用于点到点连接，即数据从一台计算机传到另一台。SSL 允许验证服务器和客户的身份；但在大多数 Internet SSL 会话中，只验证了服务器的身份。和 SSL 类似的还有 IETF（Internet 工程任务组）开发的“传输层安全”（TLS）协议。要了解 TLS 的详情，请访问 www.ietf.org/rfc/rfc2246.txt。

尽管 SSL 协议能保护通过 Internet 传输的信息，但对于信用卡号码这样的保密信息，一旦存储到商家服务器上，SSL 便不能继续提供保护。商家收到与订单附在一起的信用卡信息时，信息通常要在解密后存储到商家服务器上，直到订单生效。如果服务器不安全而且数据未加密，未经授权的第三方就可能访问这种信息。为解决这个问题，可在 Web 服务器上安装专用的 SSL PCI 卡，并用它来处理 SSL 业务。这样不仅能加快处理速度，还能为服务器腾出更多的资源去执行其他任务。要了解这些设备的更多信息，请访问 www.sonicwall.com/products/trans.asp。要了解 SSL 协议的更多信息，请访问 Netscape SSL 教程（网址是 developer.netscape.com/tech/security/ssl/protocol.html）和 Netscape Security Center（网址是 www.netscape.com/security/index.html）。

21.10.2 IPsec 和虚拟专用网络（VPN）

利用网络可连接多台计算机。局域网（LAN）连接的是物理位置接近的计算机（通常在同一幢楼里）。广域网（WAN）使用电话专线或无线电波连接位于多个地点的计算机。现在，各个组织可利用现有的 Internet 基础设施（遍布于全球的通信线缆）来创建“虚拟专用网络”（Virtual Private Network, VPN），以连接多个网络、无线用户以及其他远程用户。VPN 使用的是现成的网络基础设施，所以要比 WAN 等专用网络经济得多。通过加密技术，VPN 可通过公共网络提供与专用网络上相同的服务。

为创建 VPN，需要建立一个安全隧道，数据将通过 Internet，经过这种隧道在多个网络之间传输数据。IPsec 是用于对数据传输隧道进行保护的技术之一，可确保数据机密性和完整性，还能对用户的身份进行验证。IPsec 由 Internet 工程任务组（IETF）开发，采用公钥和对称密钥加密系统实现用户身份验证，并确保数据的完整性和机密性。该技术基于现有的标准，即信息通过 IP 在 Internet 上的两个网络之间传输。但是，用 IP 发送的信息很容易被拦截。未经授权的用户可使用许多已知技术来访问网络，比如“IP 欺骗”——攻击者伪造授权用户或主机的 IP，以访问原本受限的资源。SSL 协议在两个应用程序之间实现了安全的点到点连接；IPsec 则实现了整个网络的安全连接。IPsec 协议中往往使用 Diffie-Hellman 和 RSA 算法进行密钥交换；DES 或 3DES 用于密钥加密（具体由系统和加密需求决定）。IP 数据包先进行加密，再放到用于创建隧道的普通 IP 包中发送出去。接收者丢弃外层 IP 包，再对内层 IP 包进行解密。

VPN 的安全性依赖于 3 个基本概念：用户身份验证、数据加密后通过网络发送以及控制对企业信息的访问。为实现这 3 个安全概念，IPSec 也由 3 部分构成。“验证头”（Authentication Header, AH）为每个数据包都附加额外的信息，从而验证发送者的身份，并证明数据在传输过程中未被修改。“封装安全有效载荷”（Encapsulating Security Payload, ESP）在 IP 包从一台计算机发送到另一台时，对数据进行加密，用对称密钥防止数据被窃取。“Internet 密钥交换”（Internet Key Exchange, IKE）是 IPSec 使用的密钥交换协议，用于确定安全限制并对加密密钥进行身份验证。

VPN 在商业领域越来越流行。但是，VPN 安全很难管理。为建立 VPN，网络上的所有用户都必须安装类似软件或硬件。尽管商业伙伴可通过 VPN 方便地连接另一家公司的网络，但对特定应用程序和文件的访问只应限于特定的授权用户，不能向 VPN 上的所有用户开放。仍然要用防火墙、入侵检测软件和授权工具保护有价值的数据（参见 21.14 节）。

要想了解 IPSec 的更多信息，请访问 IPSec 开发者论坛，网址是 www.ip-sec.com。另外，还可访问 IETF 的 IPSec 工作组网站，网址是 www.ietf.org/html.charters/ipsec-charter.html。

21.11 身份验证

本章始终强调，身份验证是电子商务和移动商务安全性的一项基本要求。本节首先要介绍在网络中进行用户身份验证的一些技术，比如 Kerberos、生物测定和单一登录，然后再讨论 Microsoft Passport，它是组合了几种身份验证方法的一种技术。

21.11.1 Kerberos

防火墙不能防范来自局域网内部的威胁。内部攻击普遍存在，而且极具破坏性。例如，某个有网络访问权限的公司职员，因为受到不公正待遇，就可能在公司内部网络上搞破坏，或者盗取有价值的专利信息。据估计，公司网络的 70%~90% 攻击来自内部。Kerberos 是麻省理工学院开发的一种免费的开放源码协议。它通过密钥加密来验证网络中的用户身份，并维持网络通信的完整性和机密性。

Kerberos 系统的身份验证由一个主要 Kerberos 系统和一个辅助“票据授权服务”（Ticket-Granting Service, TGS）系统共同控制。这个系统类似于 21.3 节介绍的“密钥分发中心”（KDC）。主要 Kerberos 系统向 TGS 证明一个客户的身份；TGS 则为用户授予访问特定网络服务的权限。

网络中的每个客户都和 Kerberos 系统共享一个密钥。该密钥可由 Kerberos 系统中的多个 TGS 使用。客户首先在 Kerberos 身份验证服务器中输入一个登录名和一个密码。身份验证服务器用一个数据库维护网络中的所有客户。身份验证服务器返回一个用客户密钥（该密钥和身份验证服务器共享）来加密的“票据授予票据”（Ticket-Granting Ticket, TGT）。只有身份验证服务器和客户才知道密钥，所以只有客户才能对 TGT 解密，这样便验证了客户的身份。接着，客户系统将解密的 TGT 发送给票据授权服务，以请求一个“服务票据”。服务票据为客户授予特定网络服务的访问权限。票据有一个有效期限。过期后要续订，需重新向 TGS 申请。这样可有效地保证网络的安全性，因为对特定网络服务的访问是根据需要来授权的；一旦服务票据过期，客户就必须重新获得访问服务的授权。

21.11.2 生物测定

“生物测定”（Biometrics）是指利用指纹、眼球虹膜扫描或面部扫描等生理特征来识别用户身份。这种系统无需密码，其实密码很容易盗取。我们不是常常将密码记在一张纸上，然后将其随意放在抽屉或者钱包中吗？今天，几乎一切东西都要用到密码和 PIN——网站、网络、电子邮件、ATM 卡乃至自己的轿车，妥善保管这些密码已成为一种负担。最近，生物测定设备的成本已显著下降。集成在键盘上的指纹扫描、面部扫描和眼睛扫描设备正在逐渐取代密码，以使用户登录系统、检查电子邮件或者通过网

络访问安全信息。每个用户的虹膜扫描、面部扫描或者指纹资料都存储在一个安全数据库中。每次登录时,都要将扫描的资料与数据库中的资料比较。如果匹配,就能成功登录。两家致力于生物测定设备的公司是 IriScan(www.iriscan.com)和 Keytronic (www.keytronic.com)。

当前最流行的身份验证方式是密码。不过,我们也看到了逐渐向智能卡和生物测定转移的趋势。微软公司曾宣布会在未来的 Windows 版本中集成“物理测定应用程序编程接口”(BAPI),以便各大公司在自己的系统中集成生物测定功能。“双重身份验证”则同时采用两种方式验证用户身份,比如生物测定/智能卡和密码组合。尽管智能卡系统可能被入侵,但两种身份验证方法肯定比单独使用密码安全。

Keyware 公司发明了一种无线生物测定系统,它能在中心服务器上存储用户“声纹”。Keyware 还发明了“分层生物测定验证”(Layered Biometric Verification, LBV),它能综合运用多种生理特征,包括面部轮廓、指纹和声纹等等。利用 LBV,无线生物测定系统可将生物测定和其他身份验证方法合并到一起,比如 PIN 和 PKI 等等。

Identix 公司也在为无线事务处理提供生物测定身份验证技术。Identix 指纹扫描设备可嵌入手持设备。Identix 服务包括交易管理和内容保护。交易管理可证明一次交易确实发生,而内容保护可控制对电子文档的访问,包括限制用户下载或复制文档的能力。

目前,无线生物测定技术尚未普及。指纹扫描仪必须与安装在移动设备上的指纹阅读器配合。但是,无线设备厂家在制造指纹阅读器的问题上却犹豫不决,主要是因为这种技术的成本过高。目前已有一些笔记本电脑集成了生物测定技术,但受内存和运算能力的限制,移动电话在这方面没有多大进展。

生物测定最大的问题就是个人隐私。如果运用指纹扫描仪,公司必须在一个数据库中采集所有员工的指纹。但是,并非所有员工都乐意提供这种私人信息。另外,数据库可能被入侵,导致个人隐私不保。目前,实现了生物测定系统的大多数公司都或多或少遭遇过来自员工的阻力。

21.11.3 单一登录

要想访问不同服务器上的多个应用程序,用户必须单独提供密码,一次又一次地证明自己的身份。记住多个密码是件麻烦的事情。人们喜欢把密码记在纸上,这必然会导致严重的安全隐患。

单一登录系统则允许用户用单个密码登录一次,以后就可访问多个应用程序。当然,这个密码更需严格保护,因为一旦泄露或破解,黑客可以访问和攻击所有应用程序。

单一登录服务有三种类型:工作站登录脚本、身份验证服务器脚本和令牌。工作站登录脚本是最简单的单一登录形式。用户在其工作站登录,然后从菜单选择应用程序。工作站登录脚本将用户密码发送给应用程序服务器,并在以后访问那些应用程序时自动验证用户身份。这种脚本的安全性不够高,因为密码是以明文形式存储在 PC 上的。任何人只要能访问工作站,就能拿走用户的密码。身份验证服务器脚本则通过一台中心服务器来验证用户。服务器控制用户和应用程序的连接。这种脚本比工作站登录脚本更安全,因为密码保存在安全性优于个人 PC 的服务器上。

最高级的单一登录系统要使用基于令牌的身份验证。用户身份一经证实,就向用户颁发不可重复使用的令牌,以便他们访问特定的应用程序。登录以创建令牌的过程要通过加密或者单独一个密码加以保护,这是用户惟一需要记住或修改的密码。令牌身份验证最大的问题在于,所有应用程序都要有能力接受令牌而不是传统的登录密码。

21.11.4 Microsoft Passport

Microsoft Passport 将前面讨论的身份验证、网上购物、单一登录和其他几种技术集成到一个产品中,以便在 Internet 上的几个不同站点使用。Passport 用户只需向主 Passport 身份验证服务器登录一次。登录后,在访问每个支持 Passport 的站点时,他们会被识别为独一无二的用户。利用该技术,用户可随意检查电子邮件、和朋友聊天以及进行网上购物,无需为每个应用程序都输入密码。

登录后,身份验证信息将由 Passport 提供给合作站点。但是,实际的 Passport 密码被安全地保存一

个数据库中。Passport 使用 SSL 向中心服务器发送用户名、密码和电子钱包数据。其他站点所接收的身份验证信息采取数字密钥形式。该密码和每个用户一一对应，并可由 Passport 数据库进行验证（这类似于 PKI 基础结构）。为提高安全性，每个密钥都有一个截止时间。

密钥的有效期越短，黑客用于分析密钥或使用一个已泄密的密钥的时间越短。Microsoft Passport 还采取措施来防范野蛮破解。登录时，如果连续几次输入不正确的密码，Passport 会将账户暂时停用几分钟。这样可防止野蛮破解程序重复试验密码。

在用户计算机上，cookie 存储着加密后的个人资料。用户从 Passport 注销时，所有个人资料 cookie 都会删除。

微软的许多产品都已经或将要集成 Passport 技术。Windows XP、.NET 框架和 Hailstorm 都采用了 Microsoft Passport 技术。要想了解 Microsoft Passport 的详情以及申请免费的 Passport 账户，请访问 www.passport.com。

21.12 安全攻击

电子商务领域频繁出现网上攻击事件。拒绝服务攻击（DoS）、病毒和蠕虫已经给全世界造成了几十亿美元的损失。本节将介绍一些常见的攻击类型，并说明对信息进行保护的一些措施。

21.12.1 拒绝服务（DoS）攻击

迫使系统出现不正常行为称为“拒绝服务”（Denial of Service）。在许多 DoS 攻击中，网络资源被未经许可的通信占据，妨碍合法用户的访问。进行这种攻击时，通常要向服务器发送过量的数据包。为发起 DoS 攻击，通常要求大量计算机同时工作，尽管一些技巧性的攻击也可通过单台机器发起。DoS 攻击会导致目标计算机网络崩溃或断开连接，中断网站的正常服务，甚至造成关键系统（比如电信或飞行控制中心）关机。

e-Fact 21.5 据统计，每周约有 4 000 个站点遭受拒绝服务攻击。

在分布式拒绝服务攻击（DDoS）中，泛滥的数据包不是来自同一个地方，而是来自分散的计算机。实际上，这种攻击并不是许多人协同工作的结果。相反，它可能是个人的“杰作”。例如，他可能在许多计算机上安装病毒程序，从而非法利用这些计算机来展开攻击。DDoS 攻击很难阻止：因为不好分辨请求是来源于合法用户，还是属于攻击的一部分。另外，很难抓住发起这种攻击的罪魁祸首，因为攻击者并不是直接从自己的计算机发起攻击。

谁该为病毒和拒绝服务攻击负责？通常，我们将为此负责的人称为黑客（Hacker）或者夸客（Cracker）。虽然两个术语经常混淆，但一般地说，黑客是指有技术的程序员，而夸客只会用别人写的程序来攻击系统。根据一些说法，夸客攻击系统只是为了好玩，不会对目标系统造成太大的伤害（除了有时让对方觉得蒙羞之外）。当然，夸客如果访问或破坏私人信息和计算机，仍然是违法的。夸客的行为在客观上是恶意的，而且他们常常以攻入系统并关闭服务或窃取数据为乐。2000 年 2 月发生的分布式拒绝服务攻击关闭了大量访问量较大的网站，其中包括 Yahoo!、Bay、NN Interactive 和 Amazon。这是夸客使用一个计算机网络向网站发出大量垃圾请求，最终使网站计算机过载而引起的。尽管拒绝服务攻击的结果只是网站被迫关闭，不会影响受害者的数据，但造成的经济损失是难以估计的。例如，1996 年 8 月 6 日 eBay 网站当机 24 小时，直接导致其股价一路下滑。

21.12.2 病毒和蠕虫

病毒实际上是程序代码——通常作为附件发送，或隐藏在声音、视频和游戏中。它们附加到或者改

写其他程序,以达到复制自身的目的。病毒可能破坏文件,甚至能完全删除硬盘上的内容。Internet 问世前,病毒通过可移动磁盘上的文件和程序(比如电脑游戏)传播到计算机上。如今,病毒文件嵌入电子邮件附件、文档或程序中,并通过网络快速传播。蠕虫与病毒相似,只是它通过网络自主地传播和感染文件,无需附加到另一个程序。一旦病毒或蠕虫被释放,就会迅速散布,常常能在几分钟或者几小时内感染全世界成千上万的计算机。

计算机病毒有很多种。“临时病毒”把自己附加到特定计算机程序上。程序运行即激活病毒,终止即屏蔽病毒。更具威胁性的计算机病毒是“常驻病毒”,它们一旦载入计算机内存,就会在计算机使用时一直运行。还有一种“逻辑炸弹”病毒,满足特定条件就会发作(例如时间炸弹病毒,它们会在特定的时间或日期发作)。

“特洛伊木马”隐藏在表面无害的程序内,或者伪装成合法的程序或功能,但实际会在后台破坏计算机或网络。它的名字来源于希腊历史上著名的特洛伊战争。当时,正在和特洛伊人打仗的希腊战士隐藏在木马中,特洛伊人误将木马拉至特洛伊城内。夜深人静时,希腊战士从木马钻出来打开城门,让城外的希腊军队进入,从而一举攻破特洛伊城。特洛伊病毒很难检测,因为它们表面是合法和有用的应用程序。和木马程序联系在一起的还有“后门程序”,它们一般是常驻型病毒,使发送者能完全地、不被检测地访问受害者的计算机资源。后门程序十分危险,因为它们可能记录受害者的每一次击键操作,从而捕获密码、信用卡号码等机密信息。无论 PC 和服务端之间的连接有多么安全,只要被装上后门,就不再有任何秘密可言。根据 2000 年 6 月的新闻报道,一个木马病毒伪装成视频剪辑以邮件附件形式广为传播。木马设计宗旨是让攻击者访问受害者机器,再借助它们发起拒绝服务等形式的攻击。

两种非常著名的病毒是 1999 年 3 月发现的美丽莎(Melissa)病毒和 2000 年 5 月发现的爱虫(ILOVEYOU)病毒。这两个病毒为全球企业和个人造成了数十亿美元的损失。美丽莎通过作为电子邮件附件发送的 Microsoft Word 文档进行传播。打开附件,病毒就会发作。它能访问 Microsoft Outlook 通讯簿,并向 50 个人自动发送被感染的邮件。只要有人打开附件,病毒就能再发出 50 封邮件。进入系统后,病毒会感染将来保存的所有 Word 文档。

爱虫也以电子邮件附件的形式发送,而且貌似一封情书。邮件正文是:“Kindly check the attached love letter coming from me.”。打开附件,病毒就会访问 Microsoft Outlook 通讯簿,并向所有地址发送被感染的邮件,这使病毒快速蔓延到全世界。它能感染所有类型的文件,甚至包括系统文件。许多公司和政府机构的网络被迫关闭数日,以解决问题和清除病毒。这些病毒的泛滥,迫使人们重视邮件附件的安全性。经过这些事件,人们对操作系统和软件的安全性也提出了更高的要求。

e-Fact 21.6 爱虫病毒造成的经济损失估计高达 100~150 亿美元,而且几小时即造成惊人破坏。

病毒为什么能如此迅速地传播?一个原因是人们有强烈的好奇心打开未知来源的可执行文件。您打开过朋友发给您的声音或视频文件吗?曾经把这种文件转发给其他朋友吗?是否知道是谁制作的文件,而且其中是否存在病毒?是否打开过 ILOVEYOU 文件,并看过情书的内容?

大多数防病毒软件是被动式的。是在发现病毒后才采取措施,而不是主动防范未知病毒。一些新型病毒软件,比如 Finjan Software 公司的 SurfinGuard (www.finjan.com),则能检查附加到电子邮件的可执行文件,并在一个安全区域运行它,检测它们是否试图访问和破坏文件。要了解有关防病毒软件的更多信息,请参见补充内容“McAfee.com: 防病毒工具”

21.12.3 软件漏洞、网页毁容和网络犯罪

困扰电子商务的另一个问题是黑客可能利用软件本身的漏洞来展开破坏。除了经常更新病毒和防火墙程序之外,还应该检查联网计算机上每个程序是否存在安全漏洞。但是,由于软件产品的数量过于庞大,而且每天都有无数的漏洞被发现,所以这也是一件几乎不可能完成的任务。常见的软件漏洞是缓冲区溢出,即程序输入超出它能承受的范围。这种攻击可能导致系统崩溃,更严重的情况下,甚至允许任

意代码在计算机上运行。1993 年,一些有识之士建立了 BugTraq 网站,宗旨是以最快速度列出已知软件漏洞,并指出利用软件漏洞的具体过程,以及如何修补这些漏洞。欲知 BugTraq 的详情,请访问 www.securityfocus.com。

McAfee.com: 防病毒工具

McAfee.com 为各种用户提供了大量防病毒工具(以及其他实用程序)。这些用户的计算机可能不经常连接网络,可能一直连接到网络(比如 Internet),也可能通过无线设备(比如个人数字助理)连接到网络。

对于不经常连接网络的计算机,McAfee 准备了 VirusScan,它可以根据指示来扫描病毒,或在用户做其他事情时,进行后台病毒扫描。

对于已连接网络和能够访问 Internet 的计算机,McAfee 准备了在线的“McAfee.com Clinic”服务。用户可订阅该服务,并在当前任何一台计算机上使用网上查毒软件。和单机版 VirusScan 一样,可指定按指示来扫描文件。Clinic 服务最大的优色就是它的 ActiveShield 软件。安装好之后,ActiveShield 就能配置成扫描计算机上使用的每个文件,或者只扫描程序文件。还可让它自动检查病毒定义文件更新,并在出现更新后通知用户下载。只需单击更新通知中的超链接,即可连接到 Clinic 网站并下载更新。这样,计算机时刻都能防范最新的病毒,这是病毒保护的一个重要环节。

McAfee.com VirusScan Wireless 为 Palm、Pocket PC 和其他手持设备提供病毒保护。VirusScan Wireless 安装到用户的 PC 上。用户每次同步手持设备时,都会扫描病毒。检测到病毒,同步就会终止,并开始清除病毒。要了解 McAfee 公司的详情,请访问 www.mcafee.com。另外,也请看一看 Symantec 公司的 Norton Internetsecurity 产品(www.symantec.com)。Symantec 是全球领先的安全软件公司。其 Norton Internet Security 2003 能防黑、防毒和防止隐私泄露,尤其适用于小型公司和个人用户。

“网页毁容”是另一种流行攻击方式,即非法进入别人的网站并更改网页内容。CNN Interactive 发表过一篇题为“网上暴乱”(Insurgency on the Internet)的报告,列举了黑客惯用的攻击方式和大量黑客资料。其中最引人注目的便是一系列被“毁容”的网站的图片。1996 年,瑞典发生一起著名的网页毁容事件。当时,一些瑞典黑客将中央情报局网站(www.odci.gov/cia)的标题改为“中央愚蠢局”(Central Stupidity Agency)。似乎未能尽兴,他们还在网页上添加了淫秽内容、政治口号、给系统管理员的信以及到成人网站的链接等等。其他许多流行和大型的网站也有过被毁容的历史。对网页进行毁容似乎是今天的黑客最喜欢干的一件事情,这使人量受影响的网站(根据记录,数量超过 15000 个)被迫关闭,否则黑客们每天会对其进行骚扰。

网络犯罪可能为组织造成重大经济损失。公司无论规模大小,都要保护自己的数据、知识产权以及客户信息等等。良好的安全策略是保障数据和网络完全的关键。制定安全策略时,要擅于找出自己的漏洞以及可能存在的安全威胁。需要保护什么信息?谁是可能的攻击者?他们的目的是什么,是想窃取数据还是想破坏网络?如何响应攻击事件?要想了解有关安全和安全策略的更多信息,请访问 www.cerias.com 和 www.sans.org。在 www.baselinesoft.com 网站,提供了有关安全策略和的书籍和光盘。Baseline Software 的《Information Policies Made Easy: Version 7》含有超过 1000 个安全策略,是许多 Fortune 200 公司的必备参考书。

e-Fact 21.7 GartnerGroup 的报告指出,70%的计算机犯罪都是由不满的员工进行的。

计算机犯罪日益高涨迫使美国政府采取行动。1996 年颁布的《美国国家信息基础结构保护法》明确规定“拒绝服务”攻击和传播病毒都是犯罪行为,将课以罚金,严重时还会坐牢。要想知道美国政府惩治网络犯罪的信息,或了解美国最新公开的网络犯罪案件,请访问美国司法部网站 www.usdoj.gov/criminal/cybercrime/compcrime.html。还可访问 www.cybercrime.gov,它由美国司法部属下的 Criminal Division 维护。

设在卡耐基·梅隆大学软件工程研究所的计算机紧急反应小组(Computer Emergency Response Team,

CERT) 协调中心能对病毒和“拒绝服务”攻击做出快速反应, 还在网上公布网络安全信息, 包括如何判断系统是否已被入侵。这个站点提供了病毒和“拒绝服务”攻击事件的详细报告, 包括事件描述, 它们的影响及其对策。另外, 还报告了流行操作系统和软件所存在的安全漏洞。“CERT 安全性增强模块”是有关网络安全的极其出色的教程, 描述了用于解决网络安全问题的一些注意事项和技术。请访问 CERT 网站 (www.cert.org) 寻求更多信息。

要了解更多的网络防黑信息, 请访问 AntiOnline 网站 (www.anti-online.com), 上面提供了有关安全的新闻和信息、一个名为“向黑客宣战”的教程以及有关黑客和被黑网站记录的资料。还可访问 www.irchelp.org/irchelp/nuke, 这里提供了有关拒绝服务攻击的附加信息, 并指导您保护自己的站点。

对安全问题和方案有一个基本的认识之后, 接着让我们学习如何用 Python 保护本地计算机和网络。

21.13 运行受限 Python 代码

Python 代码独立于平台; 代码写好后, 可在几乎任何地方运行。许多程序员下载代码并通过 Python 解释器运行。但代码在本机上执行会造成安全问题, 因为代码可能非法访问本地文件或滥用计算机。要避免执行有害代码, 一个办法是在限制环境中运行代码。限制环境是一个虚拟机, 它只允许程序访问需要的资源。由于不能访问敏感资源 (比如一个硬盘驱动器或网络), 所以代码无法对这些资源造成损害。

21.13.1 rexec 模块

rexec 模块包含用于在限制环境中执行 Python 代码的 RExec 类。RExec 对象支持几个用于在限制环境中执行代码的方法, 比如 `r_eval`。在这种环境中执行时, 代码只具有对标准模块和内建 Python 函数的有限访问。对于不受信任的代码的运行环境, 程序员有完全的控制权。默认限制环境所导入的模块包括 `__builtins__` 和 `sys`。RExec 可限制对某些资源的访问, 比如磁盘或网络, 但不可限制限制环境中的代码使用的内存容量或 CPU 时间。受限代码可生成异常, 而且如果程序不捕捉和处理这些异常, 就有可能终止。因此, 程序员应在捕捉所有异常的 `try/catch` 子句中执行不受信任的代码。

21.13.2 Bastion 模块

Bastion 限制对特定对象的访问, 而不是对整个环境的访问。Bastion 对象封装一个对象, 并控制对该对象的访问。Bastion 可精确控制对象的方法; 这是通过创建 Bastion 对象时提供的一个筛选器函数来实现的。筛选器函数取得一个方法名作为参数, 如方法可以访问, 就返回 `true`。默认时, 以下划线开头的对象方法是不能访问的。

如代码试图访问受限方法, 会引发 `AttributeError` 异常。这是因为代码无法识别方法。在限制环境, 该方法根本没有定义, 所以不可访问。注意 Bastion 必须在限制环境执行, 比如在一个 RExec 环境中, 从而增强程序的安全性。

21.13.3 受限 Web 浏览器

图 21.9 演示了第 20 章展示的 Web 浏览器 (图 20.1) 的一个修改版本。修改过的浏览器检查请求的页面文件是否采用了 `.py` 扩展名。如果是, 浏览器就在一个限制环境中执行。

```
1 # Fig. 21.9: fig21_09.py
2 # Displays the contents of a file on a Web server.
3
4 from Tkinter import *
5 import tkMessageBox
```

```

6 import Pmw
7 import urllib
8 import urlparse
9 import Bastion
10 import rexec
11
12 class WebBrowser( Frame ):
13     """A simple Web browser"""
14
15     def __init__( self ):
16         """Create the Web browser GUI"""
17
18         Frame.__init__( self )
19         Pmw.initialise()
20         self.pack( expand = YES, fill = BOTH )
21         self.master.title( "Simple Web Browser" )
22         self.master.geometry( "400x300" )
23
24         self.address = Entry( self )
25         self.address.pack( fill = X, padx = 5, pady = 5 )
26         self.address.bind( "<Return>", self.getPage )
27
28         self.contents = Pmw.ScrolledText( self,
29             text_state = DISABLED )
30         self.contents.pack( expand = YES, fill = BOTH, padx = 5,
31             pady = 5 )
32
33         # create restricted environment
34         self.restricted = rexec.RExec()
35         self.module = self.restricted.add_module( "__main__" )
36         self.environment = self.module.__dict__
37
38         # add browser to environment
39         self.environment[ "browser" ] = Bastion.Bastion( self )
40
41     def setColor( self, color ):
42         """Set browser's background color"""
43
44         self.configure( background = color )
45
46     def _setColor( self, color ):
47         """Set browser's background"""
48
49         self.configure( background = color )
50
51     def setText( self, text ):
52         """Set the text of the ScrolledText component"""
53
54         self.contents.settext( text )
55
56     def runCode( self, statement ):
57         """Run a Python statement in restricted environment"""
58
59         try:
60             self.restricted.r_exec( statement ) # execute in rexec
61         except AttributeError, name:
62             tkMessageBox.showerror( "Error",
63                 "Restricted code tried to access forbidden " + \
64                 "attribute:" + str( name ) )
65
66     def getPage( self, event ):
67         """Parse the URL and addressing scheme and retrieve file"""
68
69         # parse the URL
70         myURL = event.widget.get()
71         components = urlparse.urlparse( myURL )
72         self.contents.text_state = NORMAL
73
74         # if addressing scheme not specified, use http
75         if components[ 0 ] == "":
76             myURL = "http://" + myURL

```

```

77
78     # connect and retrieve the file
79     try:
80         tempFile = urllib.urlopen( myURL ).read()
81     except IOError:
82         self.contents.setText( "Error finding file" )
83     else:
84         tempFile = tempFile.replace( "\r\n", "\n" )
85
86         if myURL[ -3: ] == ".py":
87             self.runCode( tempFile )
88         else:
89             self.contents.setText( tempFile ) # show results
90
91     self.contents.text_state = DISABLED
92
93 def main():
94     WebBrowser().mainloop()
95
96 if __name__ == "__main__":
97     main()

```

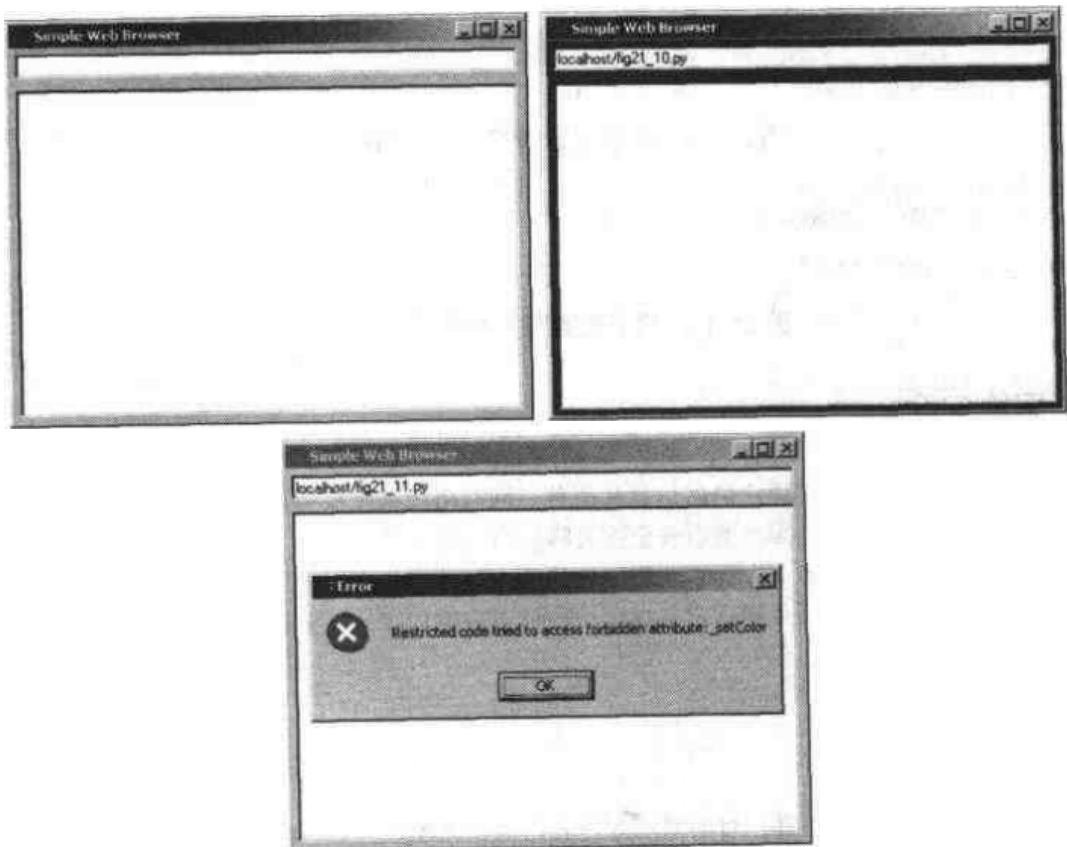


图 21.9 受限 Web 浏览器

第 34 行创建 RExec 类的一个对象。第 35 行获得环境的 `__main__` 模块。对象定义一个环境，其中包含一系列由可访问的模块和内建函数（例如 `raw_input` 或 `abs`）。它有自己的环境，包括一系列可访问的模块和内建方法。其中，`add_module` 方法在限制环境所允许的模块列表中添加一个新模块，返回的是对那个模块的一个引用。如果环境已经允许访问该模块，`add_module` 就只返回对指定模块的一个引用。`add_module` 不将模块导入限制环境；只是修改受限代码可以导入的模块的列表。

第 36 行获得对字典 `__dict__` 的引用，字典中包含用于限制环境的模块全局绑定。Bastion 模块封装一个 Web 浏览器组件，并把它添加到限制环境的模块全局命名空间（第 39 行）。受限代码现在可访问和操纵 Web 浏览器组件。使用 Bastion 封装 Web 浏览器组件，程序就可限制代码访问浏览器的方式。默认情

况下，代码不能访问一个 Bastion 对象的数据成员，也不能访问任何以下划线（_）开头的方法。代码可访问不以下划线开头的方法。

为演示代码执行，第 41~49 行为 WebBrowser 添加两个方法。setColor（第 41~44 行）和 _setColor 方法（第 46~49 行）设置 WebBrowser 的前景颜色。默认情况下，代码不能访问由 Bastion 封装的浏览器对象的 _setColor 方法。

请求 Python 程序会导致调用 runCode 方法（第 56~64 行）。try 语句（第 59~64 行）在限制环境中执行 Python 程序。如程序试图访问 _setColor 方法，就会捕捉 AttributeError 异常，并用一个消息框显示警告。在更实际的例子中，try 语句可包括一个空的 except 子句。该 except 子句会捕捉代码所引发的其他异常，不管这些错误是有意还是无意造成的。程序可向错误文件写一条消息，或者在对话框/控制台中显示一条消息。这样做的主要目的是防止所下载的代码中的一个错误造成程序终止。

图 21.9 的屏幕截图演示了运行图 21.10 和图 21.11 的代码的结果。第一幅屏幕截图是浏览器的初始状态。第二幅截图是运行图 21.10 的代码的结果。浏览器将背景色变成蓝色。最后一幅截图演示图 21.9 的代码通过受限的 _setColor 方法更改颜色时的结果。

```
1 # Fig. 21.10: fig21_10.py
2 # Calls method setColor to set background color.
3
4 browser.setColor( "blue" )
```

图 21.10 用于设置背景色的 setColor

```
1 # Fig. 21.11: fig21_11.py
2 # Method _setColor raises attribute error in restricted enviroment.
3
4 browser._setColor( "red" )
```

图 21.11 用于设置背景色的 _setColor

21.14 网络安全

网络安全的目的是允许授权用户访问信息和服务，同时防止未授权用户访问（并可能破坏）网络。网络安全和网络性能需要折衷，因为增强安全性会降低网络效率。

本节讨论了网络安全性的各个方面。我们要讨论防火墙，它将未授权用户阻挡在网络外部；要讨论授权服务器，它允许用户根据一系列预定义规则访问特定应用程序；还要讨论入侵检测系统，它能主动监视网络，防止入侵和攻击。

21.14.1 防火墙

防火墙是基本的网络安全工具，目的是将入侵者阻挡在 LAN 外部。例如，大多数公司都有内部网络，允许员工共享和访问公司信息。每个 LAN 都可通过网关连接 Internet，而网关通常包含防火墙。多年来，最大的安全威胁之一来源于防火墙内部的员工。现在，公司严重地依赖 Internet，越来越多的安全威胁来源于防火墙外部——即通过 Internet 连接公司网络的成千上万的人。防火墙是进出于 LAN 的数据的安全屏障。防火墙可禁止所有没有明确允许的数据，也可允许所有没有明确禁止的数据。具体选择哪一种，要取决于网络安全管理员，而且应依据对安全和功能的要求。

主要有两种类型的防火墙：“包过滤防火墙”和“应用程序级网关”。包过滤防火墙检查从 LAN 外部发送的任何数据，自动拒绝具有本地网络地址的所有数据包。例如，假定网络外的黑客获得网络内部的一台计算机的地址，并试图通过防火墙传入一个有害的数据包，包过滤防火墙就会拒绝该数据包，因其使用内部地址，但又来源于网络外部。这种防火墙的问题在于，它们只检查数据包的来源；而不检查实际数据。所以，黑客可在授权用户的计算机上安装有害的病毒，并在用户不知道的前提下访问网络。

应用程序级网关的宗旨则是筛选实际数据。只有认为数据本身是安全的，才发送给目标接收者。

防火墙是改进小型网络安全性的最简单、最有效的方式。通常，小型公司或家庭用户通过持久性连接（比如 DSL 线路）上网时，不会采用非常强大的安全机制。所以，他们的计算机成为黑客发起拒绝服务攻击或者窃取信息的首要目标。所以，有必要保证所有上网计算机都具有一定级别的安全性。目前有大量防火墙软件可供选择。另外，Windows XP 集成一个简单的“Internet 连接防火墙”，尤其适合小型公司和家庭用户使用。

Air Gap 技术是一种网络安全方案，它弥补了防火墙的不足。从外部访问内部网络资源时，它能保护私有数据的安全。Air Gap 使内部网络与外部网络分离，组织要确定哪些信息可由外部用户使用。Whale Communications 建立了一种 e-Gap 系统，它包括两台服务器和一个 Memory Bank。Memory Bank 不运行操作系统；所以，黑客无法利用操作系统已知的缺陷来访问网络信息。

Air Gap 技术不允许外部用户查看网络结构，防止黑客搜索结构以查找漏洞或特定的数据。e-Gap Web Shuttle 特性通过限制系统的 Back Office，从而实现安全的外部访问。Back Office 控制着整个公司最敏感的信息以及基于 IT 的商业过程。用户要想访问隐藏在 Air Gap 之后的网络，必须先通过 Air Gap 处的身份验证服务器。获得授权的用户可享受“单一登录”功能，即使用一个登录密码即可访问网络中允许的所有区域。

e-Gap Secure File Shuttle 特性负责将文件传入、传出网络。每个文件都在 Air Gap 之后进行检查。如果认为文件是安全的，就由 File Shuttle 送入网络。

Air Gap 技术由大量电子商务公司使用，便于它们的客户和合作伙伴自动访问信息，从而节省了管理成本。存储着大量机密信息的许多军事、宇航和政府机构也在使用这种技术。

SANS 协会：安全研究和教育

“系统管理、联网和安全协会”（System Administration, Networking and Security Institute, SANS）成立于 1989 年，是一个著名的安全研究和教育组织，有超过 96000 名成员（www.sans.org）。SANS 出售安全培训、认证程序和出版物。该组织还提供几项免费的公益服务，包括安全警告和新闻。

SANS 每年发布“Roadmap to Security Tools and Services Poster”产品，其中的信息涉及关键安全技术，致力于发展不同技术的安全厂商列表以及到其他安全信息的 URL。同时还指导如何订购约 20 份白皮书。要订购这个产品，并申请获得技术白皮书拷贝，请访问 www.sans.org/tools.htm。

SANS Information Security Reading Room 是一个出色的安全信息资源。该网站提供数百篇文章和案例分析，按安全主题进行组织。这些主题包括身份验证、防攻击、入侵检测、代码安全防护、安全标准等。欲知详情，请访问 www.sans.org/infosecFAQ/index.htm。

SANS 提供 3 份免费电子刊物。其中，“SANS NewsBites”主要列出关键的新闻文章（附简要总结），并给出文章链接。请访问 www.sans.org/newlook/digests/newsbites.htm 以查看最新刊物、查询过刊或者免费订阅。“Security Alert Consensus, SAC”是每周发行的电子刊物，汇总一周的安全警告和对策。订户可选择接收有关特定操作系统的信息。“SANS Windows Security Digest”列出 Windows NT 安全更新、威胁和 Bug。要订阅任何 SANS 电子刊物，请访问 www.sans.org/sansnews。

SANS 全球事件分析中心（Global Incident Analysis Center, GIAC）记录最新的攻击，并对每种攻击进行分析。网络和系统管理员可利用这种信息防卫自己的网络和系统。在 www.sans.org/giac.htm 和 www.incidents.org，各种报告会快速公布。

21.14.2 入侵检测系统

黑客进入防火墙该怎么办？怎样知道入侵者渗透了防火墙？另外，如何未经授权的员工正在访问受限的应用程序？入侵检测系统监视网络和应用程序日志文件（包括有关文件的信息，比如访问者是谁，是在什么时候访问的）。一旦入侵者进入网络或访问未经许可的应用程序，系统就会检测到入侵，停止会话，并向系统管理员发出警报通知。

基于主机的入侵检测系统可监视系统应用程序日志。例如，可用它们扫描木马。基于网络的入侵检测软件可监视网络通信，找出不寻常的访问模式。这些模式可能意味着有人发起 DoS 攻击，或者未经许可的用户试图进入网络。然后，管理员可检查日志文件，判断是否真的发生入侵。如果是，就依据日志信息跟踪罪犯。许多公司都提供了入侵检测产品，包括思科（www.cisco.com/warp/public/cc/pd/sqsw/sqidsz）、惠普（www.hp.com/security/home.html）和赛门铁克（www.symantec.com）。

卡耐基·梅隆大学软件工程研究所正在开发一种“Operationally Critical Threat, Asset and Vulnerability Evaluation”（OCTAVESM）方法，用于评估系统面临的安全威胁。OCTAVE 包括三个阶段：建立威胁原型、识别弱点和漏洞以及开发安全方案和计划。在第一阶段，组织要定好自己的重要信息和资产，评估保护它们所需的安全级别。在第二阶段，要检查系统的弱点和漏洞，它们可能造成对有价值的数据的损害。第三阶段是在 OCTAVE 指定的 3~5 名安全专家的帮助下，开发一个安全策略。这种方式最大的特点在于，计算机用户不仅能获得对其系统的专业分析，还得亲自参与为关键信息制定优先级的过程。

21.15 隐写术

“隐写术”（Steganography）是指将信息隐藏在其他信息中的一种技术。通俗地说，就是“密写”。类似密码系统，隐写术在古代就开始运用。利用隐写术，可将信息（比如一条消息或者一张图片）隐藏到其他图片、消息甚至声音剪辑中。隐写术利用了数字文件、图片或者可移动磁盘中的多余空间。一条消息想秘密发送出去，可把它隐藏在另一条消息中，这样只有目标接收者才能读到这条消息。例如，假设您想指示经纪人购买某只股票，但消息又必须通过不安全的渠道传输，就可将消息写为“BURIED UNDER YARD”。因为您事先和经纪人达成一致，真正的消息将隐藏在每个单词的首字母里，所以经纪人一看就明白这是“BUY”。

隐写术流行的一种应用是“数字水印”，主要用于保护知识产权。图 21.12 展示的是一个传统水印。数字水印也许可见，也许不可见。它常常是公司徽标、版权声明或其他表明所有者身份的标记或消息。例如，文件的所有者可在法庭上出示隐藏水印，证明这份文件是被人偷窃的。



图 21.12 传统水印

数字水印对电子商务有重要价值。以唱片录音工业为例，音乐出版商担心 MP3 技术造成盗版歌曲、唱片集的泛滥。所以，他们一般不愿意把音乐放到网上；数字内容太容易拷贝。另外，CD-ROM 是数字化的，也很容易通过 Web 上传和共享音乐。使用数字水印，音乐出版商可对歌曲的一部分进行让人难以察觉的改动（在人耳听不到的频率范围上），以证明一份音乐产品是否盗版。微软为数字声音研制了一种水印系统，并集成到最新的 Windows Media Player 软件中。利用这种系统，可在歌曲中嵌入像许可信息这样的数据；而 Media Player 不会播放含有无效信息的文件。

e-Fact 21.8 全球唱片录音工业每年因为盗版而损失约 50 亿美元。

Blue Spike 公司的 Giovanni 数字水印软件使用加密密钥生成隐写式数字水印，并将它嵌入数字音乐和图片（图 21.13）。水印可证明所有权归属，可帮助数字内容出版商保护有版权的产品。不知道嵌入机制，就无法探测水印的存在，因而也无法识别和移除水印。水印可随机放在产品中的任何位置。

Giovanni 结合了密码系统和隐写术。它根据加密算法和媒体文件内容生成一个密钥。然后，用密钥将水印放在文件中，以后也要依据这个密钥进行解码。软件可识别图像或声音文件中令人难以察觉的区域，将数字水印无声无息地嵌入这些区域。如强行移除水印，内容也会被破坏。

目前，数字水印功能已嵌入许多图像编辑软件，其中包括 Adobe Photoshop 7.0 (www.adobe.com)。提供数字水印方案的公司则包括 Digimarc(www.digimarc.com)和 Cognicity 公司(www.cognicity.com)。

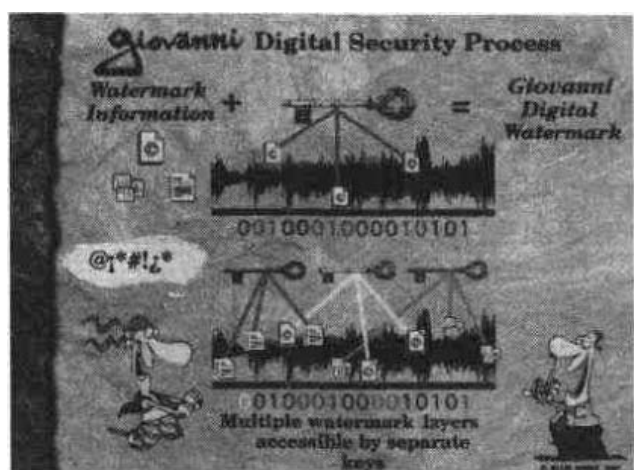


图 21.13 隐写术示例: Blue Spike 的 Giovanni 数字水印处理过程

第 22 章 数据结构

学习目标

- 使用自引用类和递归生成链接数据结构
- 创建和操纵动态数据结构，如链表、队列、堆栈和二叉树
- 理解链接数据的几种重要应用
- 理解如何通过合成创建可重用的数据结构

22.1 概述

本章介绍数据结构的常规主题，它们是 Python 的一些基本数据类型（比如列表、元组和字典）的基础。链表是数据项的集合，这些数据项“排成一列”，可在链表的任何地方插入和移除。堆栈在编译器和操作系统中非常重要，只能在堆栈头尾进行插入和移除。队列类似一队正在排队的人，只能在队尾插入，从队头移除。二叉树用于实现数据高速搜索和排序，可有效消除重复数据项，可表示文件系统目录，并可將表达式编译成机器语言。

本章讨论了主要数据结构类型，并以实例创建和操纵这些数据结构。我们使用类和合成来创建和打包数据结构，以改善重用性和可维护性。

尽管基本的 Python 列表可作为堆栈和队列使用，但本章通过“从头”学习和创建这些结构，可为更高级的计算机课程打下坚实基础。本章的例子非常实用，可在更高级的课程以及行业应用程序中运用。

22.2 自引用类

自引用类包含一个成员，它引用的是同一个类的对象。下面来看一个 Node 类，它包括两个 data 数据成员，即成员 data 和引用成员 nextNode。nextNode 成员引用 Node 类的一个对象，即与当前定义的属于同一类的对象，这正是“自引用类”一词的来历。nextNode 成员称为“链接”；也就是说，可用 nextNode 将 Node 类的一个对象“绑定”到相同类型的另一个对象。Node 类还有两个方法：构造函数接收一个值来初始化 data 成员；__str__ 方法则将节点的数据表示成一个字符串。

自引用类的对象可链接到一起构成有用的数据结构，比如链表、队列、堆栈和树。图 22.1 展示了如何将自引用类的 3 个对象链接到一起构成列表。注意斜线代表一个 None 引用，它放在第 3 个对象的链接成员中，表明不再引用另一个对象。斜线只是为了叙述方便，和 Python 的反斜杠字符没有任何关系。None 引用通常表明数据结构的结束。

常见编程错误 22.1 列表最后一个节点不设置 None 链接是逻辑错误。

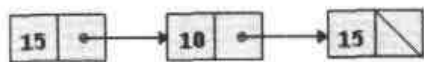


图 22.1 自引用类的 3 个链接对象

22.3 链表

“链表”（Linked List）是自引用类的对象的线性集合。这些对象称为“节点”（Node），节点通过“引用链接”而连接到一起。这正是“链表”一词的来历。要访问链表，需引用列表的第一个节点（链首节点）。后续节点则通过存储在每个节点的引用链接来访问。按照惯例，链尾节点的链接要设为 None，它

标记链表结束。数据在链表中动态存储——每个节点都是根据需要而创建的。节点可包含任意类型的数据，其中包括其他类的对象。堆栈和队列也是线性数据结构，而且正如以后要讲到的，它们只是链表的功能限制版本。“树”则属于非线性数据结构。

可在列表的恰当位置插入每个新元素，从而保持链表的排序顺序。现有的列表元素无需移动。

性能提示 22.1 在普通排序列表中插入和删除较花时间，因为在插入或删除的元素之后，所有元素都必须相应地移位。相反，在排序链表中插入和删除最多只需修改3次引用链接。

链表节点通常不在内存中连续存储。但从逻辑上说，链表各节点在表面上是连续的。图 22.2 展示了含有几个节点的一个链表。

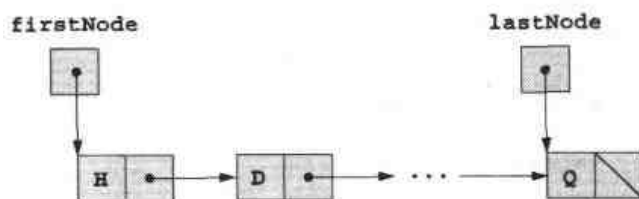


图 22.2 链表示意图

图 22.3 包含了 Node 和 List 类的定义。图 22.4 的程序为用户提供 5 个选项来操纵 List 类的一个对象：

1. 在链首插入一个值 (insertAtFront 方法)。
2. 在链尾插入一个值 (insertAtBack 方法)。
3. 从链首删除一个值 (removeFromFront 方法)。
4. 从链尾删除一个值 (removeFromBack)。
5. 终止列表处理。

```

1 # Fig. 22.3: ListModule.py
2 # Classes List and Node definitions.
3
4 class Node:
5     """Single node in a data structure"""
6
7     def __init__( self, data ):
8         """Node constructor"""
9
10        self._data = data
11        self._nextNode = None
12
13    def __str__( self ):
14        """Node data representation"""
15
16        return str( self._data )
17
18    class List:
19        """Linked list"""
20
21        def __init__( self ):
22            """List constructor"""
23
24            self._firstNode = None
25            self._lastNode = None
26
27        def __str__( self ):
28            """List string representation"""
29
30            if self.isEmpty():
31                return "empty"
32
33            currentNode = self._firstNode
34            output = []
35
```

```

36     while currentNode is not None:
37         output.append( str( currentNode._data ) )
38         currentNode = currentNode._nextNode
39
40     return " ".join( output )
41
42 def insertAtFront( self, value ):
43     """Insert node at front of list"""
44
45     newNode = Node( value )
46
47     if self.isEmpty(): # List is empty
48         self._firstNode = self._lastNode = newNode
49     else: # List is not empty
50         newNode._nextNode = self._firstNode
51         self._firstNode = newNode
52
53 def insertAtBack( self, value ):
54     """Insert node at back of list"""
55
56     newNode = Node( value )
57
58     if self.isEmpty(): # List is empty
59         self._firstNode = self._lastNode = newNode
60     else: # List is not empty
61         self._lastNode._nextNode = newNode
62         self._lastNode = newNode
63
64 def removeFromFront( self ):
65     """Delete node from front of list"""
66
67     if self.isEmpty(): # raise exception on empty list
68         raise IndexError, "remove from empty list"
69
70     tempNode = self._firstNode
71
72     if self._firstNode is self._lastNode: # one node in list
73         self._firstNode = self._lastNode = None
74     else:
75         self._firstNode = self._firstNode._nextNode
76
77     return tempNode
78
79 def removeFromBack( self ):
80     """Delete node from back of list"""
81
82     if self.isEmpty(): # raise exception on empty list
83         raise IndexError, "remove from empty list"
84
85     tempNode = self._lastNode
86
87     if self._firstNode is self._lastNode: # one node in list
88         self._firstNode = self._lastNode = None
89     else:
90         currentNode = self._firstNode
91
92         # locate second-to-last node
93         while currentNode._nextNode is not self._lastNode:
94             currentNode = currentNode._nextNode
95
96         currentNode._nextNode = None
97         self._lastNode = currentNode
98
99     return tempNode
100
101 def isEmpty( self ):
102     """Returns true if List is empty"""
103
104     return self._firstNode is None

```

图 22.3 链表表示 - ListModule.py

```
1 # Fig. 22.4: fig22_04.py
2 # Driver to test class List.
3
4 from ListModule import List
5
6 # instructions for user
7 instructions = """Enter one of the following:
8     1 to insert at beginning of list
9     2 to insert at end of list
10    3 to delete from beginning of list
11    4 to delete from end of list
12    5 to end list processing\n"""
13
14 listObject = List() # create empty List
15 print instructions # print instructions
16
17 # obtain user choice until user chooses to quit (choice 5)
18 while 1:
19
20     choice = raw_input("\n? ")
21
22     # insert at front
23     if choice == "1":
24         listObject.insertAtFront( raw_input( "Enter value: " ) )
25         print "The list is: ", listObject
26
27     # insert at end
28     elif choice == "2":
29         listObject.insertAtBack( raw_input( "Enter value: " ) )
30         print "The list is: ", listObject
31
32     # delete from front
33     elif choice == "3":
34
35         try:
36             value = listObject.removeFromFront()
37         except IndexError, message:
38             print "Failed to remove:", message
39         else:
40             print value, "removed from list"
41             print "The list is: ", listObject
42
43     # delete from end
44     elif choice == "4":
45
46         try:
47             value = listObject.removeFromBack()
48         except IndexError, message:
49             print "Failed to remove:", message
50         else:
51             print value, "removed from list"
52             print "The list is: ", listObject
53
54     # exit
55     elif choice == "5":
56         break # terminate while loop
57
58     # invalid choice
59     else:
60         print "Invalid choice:", choice
61         print instructions
62
63 print "End list test\n"
```



```

Enter one of the following:
 1 to insert at beginning of list
 2 to insert at end of list
 3 to delete from beginning of list
 4 to delete from end of list
 5 to end list processing

? 2
Enter value: 2
The list is: 2

? 2
Enter value: 3
The list is: 2 3

? 1
Enter value: 1
The list is: 1 2 3

? 3
1 removed from list
The list is: 2 3

? 4
3 removed from list
The list is: 2

? 3
2 removed from list
The list is: empty

? 4
Failed to remove: remove from empty list

? 5
End list test

```

图 22.4 链表表示 - fig22_04.py

图 22.3 由两个类构成，即 Node 和 List。在 List 类的每个对象中都封装了由 Node 构成的链表。Node 的 `_nextNode` 成员存储着对链表下一个 Node 类对象的引用。`__str__` 方法（第 13~16 行）返回 Node 的数据的字符串表示。

List 类包括 `_firstNode`（引用 List 的第一个 Node）和 `_lastNode`（引用 List 的最后一个 Node）这两个成员。构造函数将两个链接都初始化为 None（第 24~25 行）。List 的主要方法包括 `insertAtFront`（第 42~51 行）、`insertAtBack`（第 53~62 行）、`removeFromFront`（第 64~77 行）以及 `removeFromBack`（第 79~99 行）。

`isEmpty` 方法（第 101~104 行）称为“断言方法”，它并不改动 List，只是判断 List 是否为空（也就是对 List 的第一个 Node 的引用是否为 None）。如 List 为空，返回 1；否则返回 0。`__str__` 方法（第 27~40 行）显示 List 内容。

良好编程习惯 22.1 为新节点的链接成员指派 None 值。

软件工程知识 22.1 由于 Python 采用引用计数机制，所以一旦不再存在对 List 类的对象的引用，List 及其引用的所有 Node 都会被销毁（只要不存在其他 Node 引用）。但在没有引用计数机制或采用自动垃圾回收机制的语言中（比如 C 或 C++），需要人工删除对这些对象的所有引用，再销毁它们（比如用一个析构方法）。

接着详细讨论 List 类的每个方法。其中，`insertAtFront` 方法（图 22.3 的第 42~51 行）在列表头放入一个新节点。该方法由几个步骤组成（同时参见图 22.5）。

1. 新建 Node 类的一个对象，将引用存储到变量 `newNode` 中（第 45 行）。
2. 如列表为空，将 `_firstNode` 和 `lastNode` 都设为 `newNode`（第 48 行）。
3. 如列表非空，新节点将进入列表，做法是将 `_firstNode` 引用的节点指派给 `newNode._nextNode`，使新节点链接原来的链首节点，再将 `_firstNode` 设置成 `newNode` 引用（第 50~51 行）。

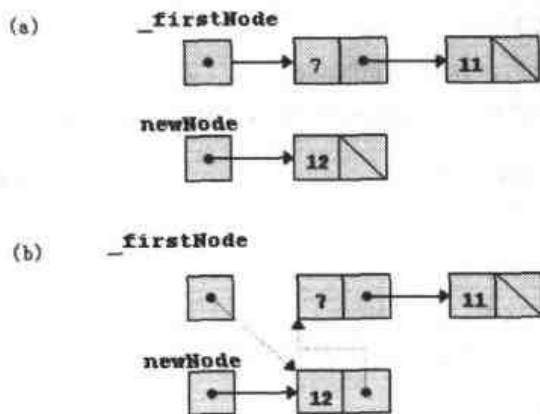


图 22.5 insertAtFront 操作示意图

在图 22.5 中，(a)展示了在 `insertAtFront` 操作期间，并在新节点进入列表之前，列表与新节点的样子。(b)中的虚线箭头对应于 `insertAtFront` 操作的步骤 3，它使包含 12 的节点成为新的链首节点。

`insertAtBack` 方法（图 22.3 第 53~62 行）在链尾放入一个新节点。该方法由几个步骤组成（同时参见图 22.6）：

1. 新建包含 `value` 的一个列表节点，并将节点指派给 `newNode` 引用（第 56 行）。
2. 如果列表为空，将 `_firstNode` 和 `_lastNode` 都设为 `newNode`（第 59 行）。
3. 如列表不为空，新节点进入列表，做法是将 `newNode` 引用的节点指派给 `_lastNode._nextNode`，使原来的链尾节点现在链接新节点，再将 `_lastNode` 设置成 `newNode` 引用（第 61~62 行）。

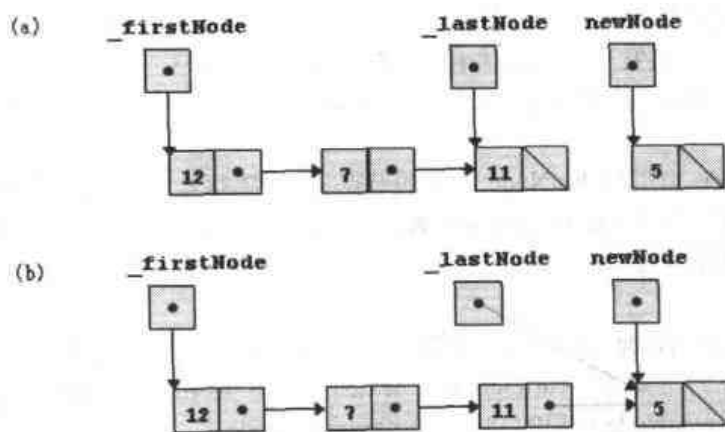


图 22.6 insertAtBack 操作示意图

在图 22.6 中，(a)展示了在操作期间，并在方法插入新节点之前，列表和新节点的样子。(b)中的虚线箭头对应于 `insertAtBack` 操作的步骤 3，它使一个新节点添加到非空列表的尾部。

`removeFromFront` 方法（图 22.3 的第 64~77 行）删除链首节点，并返回对那个节点的一个引用。如果试图从空列表删除节点，方法会引发 `IndexError` 异常。方法由以下几个步骤组成：

1. 如果列表为空，引发一个 `IndexError` 异常（第 67~68 行）。
2. 将 `_firstNode` 引用的节点指派给名为 `tempNode` 的一个新引用（第 70 行）。方法最终返回该引

用。

3. 如果 `_firstNode` 引用与 `_lastNode` 相同的节点，即列表在进行删除操作前只含有一个元素（第 72 行），就将 `_firstNode` 和 `_lastNode` 都设为 `None`（第 73 行），这样便得到一个空列表。
4. 如果在删除操作之前，列表含有多个节点，就保持 `_lastNode` 不变，但将 `_firstNode` 设为由 `_firstNode._nextNode` 引用的节点（第 75 行）。也就是修改 `_firstNode`，使其引用删除之前的第二个节点（现在变成新的链首节点）。
5. 所有引用都处理完毕后，返回 `tempNode` 引用（第 77 行）。

在图 22.7 中，(a)显示了删除操作之前的列表情况，(b)显示了实际的引用操作过程。

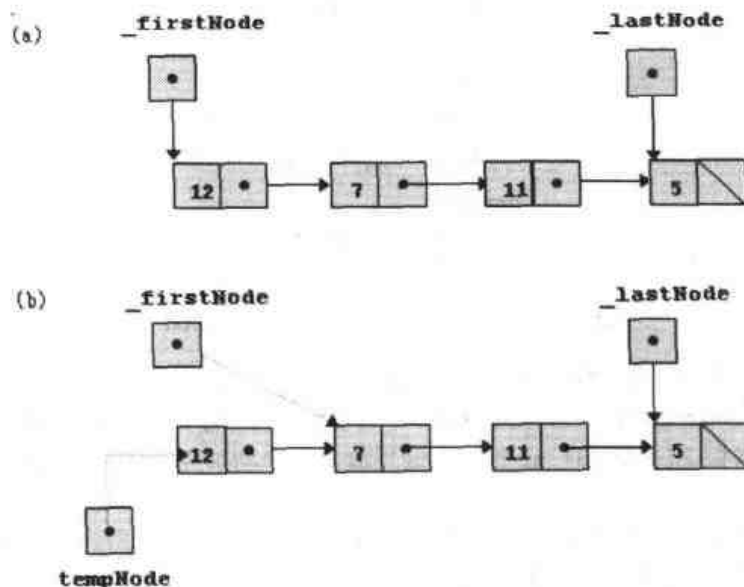


图 22.7 `removeFromFront` 操作示意图

`removeFromBack` 方法（图 22.3 的第 79~99 行）删除链尾节点，并返回那个节点的值。如果试图从空列表删除节点，方法会引发 `IndexError` 异常。方法由以下几个步骤组成：

1. 如果列表为空，引发一个 `IndexError` 异常（第 82~83 行）。
2. 将 `_lastNode` 引用的节点指派给名为 `tempNode` 的一个新引用（第 85 行）。方法最终返回该引用。
3. 如果 `_firstNode` 引用与 `_lastNode` 相同的节点（第 87 行），即列表在进行删除操作前只含有一个元素，就将 `_firstNode` 和 `_lastNode` 都设为 `None`（第 88 行），这样便得到一个空列表。
4. 如果在进行删除操作前，列表含有多个节点，就将 `_firstNode` 引用的节点指派给 `currentNode`（第 90 行）。
5. 接着利用 `currentNode` 遍历列表，直到它引用倒数第二个节点。这个过程用 `while` 循环完成（第 93~94 行）。如果 `currentNode._nextNode` 不是由 `_lastNode` 引用的节点，就将 `currentNode._nextNode` 引用的节点指派给 `currentNode`。
6. 将 `currentNode` 的 `_nextNode` 设为 `None`，再将 `currentNode` 指派给 `_lastNode`（第 96~97 行）。
7. 所有引用都处理完毕后，返回 `tempNode` 引用（第 99 行）。

在图 22.8 中，(a)展示了删除操作之前的列表。(b)展示了实际的引用处理过程。`List` 的 `__str__` 方法（图 22.3 的第 27~40 行）首先判断列表是否为空。如果是，方法返回 `"empty"`。否则，它返回一个字符串，其中包含每个节点的数据。方法将 `currentNode` 初始化成 `firstNode` 引用，并将变量 `output` 初始化成空列表。在 `currentNode` 不是 `None` 的前提下，将 `currentNode._data` 的字符串表示添加到列表，并将 `currentNode` 设为 `currentNode._nextNode` 引用。最后一个节点的数据添加到输出列表后，`__str__` 方法返回对输出列表

调用字符串方法 `join` 的结果。这会创建一个字符串，其中包含每个节点的数据，中间用空格（" "）字符分隔。注意假如链尾节点的链接不是 `None`，字符串创建算法会错误地越过列表尾。链表、堆栈和队列采用完全相同的字符串创建算法。

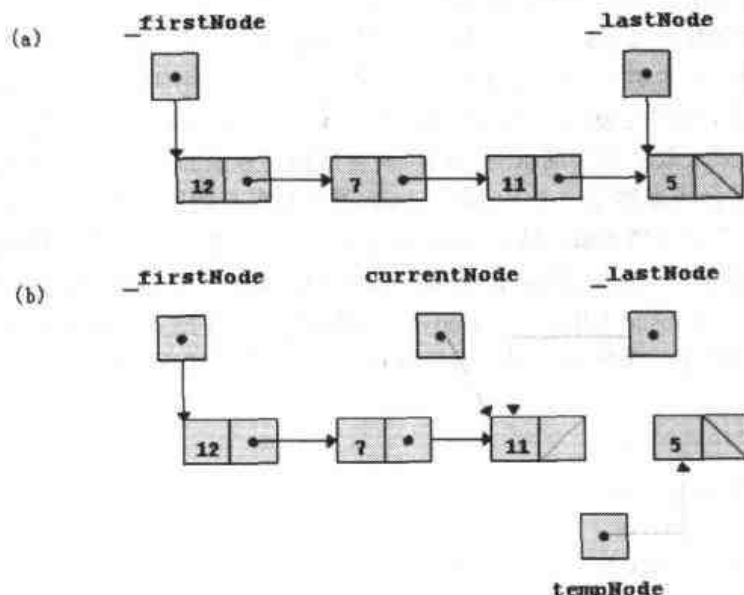


图 22.8 `removeFromBack` 操作示意图

前面讨论的是“单向链表”，最开始是对链首节点的引用，然后每个节点都顺序引用下一个节点。遇到引用成员为 `None` 的节点，就终止链表。单向链表只能向一个方向遍历。

“循环单向链表”最开始是对链首节点的引用，然后每个节点都引用下一个节点。但是，“链尾节点”不是引用 `None`，而是引用链首节点，由此完成一个“循环”。

“双向链表”允许向前和向后遍历。这种列表通常用两个“链首引用”实现，一个引用链首节点，实现从前向后遍历；另一个引用链尾元素，实现从后向前遍历。每个节点都在正方向上，向前引用列表的下一个节点；也在反方向上，向后引用列表的下一个节点。例如，假定链表包含按字母顺序排列的电话目录，要搜索的人名首字母在字母表中靠前，可从链首开始搜索；首字母在字母表中靠后，就可从链尾开始搜索。

在“循环双向链表”中，链尾节点的向前引用指向链首节点，首节点的向后引用则指向链尾节点，由此完成一个“循环”。

22.4 堆栈

“堆栈”（Stack）是链表的功能限制版本，只能在顶部添加或删除节点。换言之，堆栈是“后入先出”（LIFO）数据结构。堆栈最后一个节点的链接成员设为 `None`，标记堆栈的底部。

常见编程错误 22.2 不将堆栈底部节点的链接设为 `None` 是逻辑错误。

用于操纵堆栈的主要方法是 `push` 和 `pop`。`push` 在堆栈顶部添加一个新节点，`pop` 则从顶部移除一个节点，并将它的值返回给调用者。如堆栈为空，方法会引发 `IndexError` 异常。

堆栈有许多有趣的应用。例如进行函数调用时，被调用的函数必须知道如何返回到它的调用者，所以返回地址被压入堆栈。如果发生一系列函数调用，就按后入先出顺序将一系列返回地址压入堆栈，使每个函数都能返回至它的调用者。堆栈支持递归函数调用，采取的方式和普通非递归调用一样。

每次调用函数时，都会在堆栈内为局部变量创建空间。一旦函数返回至调用者，或引发异常，就会为每个局部对象调用析构函数（如果有的话）。为那个函数的局部变量保留的空间会弹出堆栈，程序不能再使用那些变量。编译器在表达式求值以及生成机器语言码的过程中，会使用堆栈。

下面利用列表和堆栈的密切关系来实现一个堆栈类。这主要是通过重用一個列表类来完成的。我们通过“合成”来实现堆栈类，即堆栈类的数据成员引用了 List 类的一个对象。还可通过“继承”来实现堆栈，也就是从 List 类派生出 Stack 类。所以，我们为 ListModule 模块（图 22.3）中的数据采用单下划线命名方案。如有必要，派生类可访问列表的数据，但类的客户使用所提供的方法。通过合成来实现堆栈的好处在于，Stack 类只提供了进行堆栈操作所需的方法（即 push 和 pop）。类不会暴露除堆栈操作之外的其方法。相反，如果采取从基类继承的方式，就会暴露出多余的方法，比如 removeFromBack。

图 22.9 的程序通过合成来创建一个 Stack。Stack 类具有私有数据成员 _stackList，它是图 22.3 定义的 List 类的一个对象。我们希望 Stack 提供 push 和 pop 方法。实现 Stack 的方法时，要让每个方法都调用 _stackList 对象的合适方法；也就是说，push 调用 insertAtFront，而 pop 调用 removeFromFront。当然，List 类还包含其他方法（即 insertAtBack 和 removeFromBack），它们是 Stack 类不会调用的。图 22.10 的 driver 程序使用 Stack 实例化一个堆栈对象，然后将 0~3 的整数压栈后出栈。

```

1 # Fig. 22.9: StackModule.py
2 # Class Stack definition.
3
4 from ListModule import List
5
6 class Stack ( List ):
7     """Stack composed from linked list"""
8
9     def __init__( self ):
10         """Stack constructor"""
11
12         self._stackList = List()
13
14     def __str__( self ):
15         """Stack string representation"""
16
17         return str( self._stackList )
18
19     def push( self, element ):
20         """Push data onto stack"""
21
22         self._stackList.insertAtFront( element )
23
24     def pop( self ):
25         """Pop data from stack"""
26
27         return self._stackList.removeFromFront()
28
29     def isEmpty( self ):
30         """Return 1 if Stack is empty"""
31
32         return self._stackList.isEmpty()

```

图 22.9 堆栈实现 - StackModule.py

```

1 # Fig. 22.10: fig22_010.py
2 # Driver to test class Stack.
3
4 from StackModule import Stack
5
6 stack = Stack()
7
8 print "Processing a Stack"
9
10 for i in range( 4 ):
11     stack.push( i )
12     print "The stack is:", stack
13

```

```

14 while not stack.isEmpty():
15     pop = stack.pop()
16     print pop, "popped from stack"
17     print "The stack is:", stack

```

```

Processing a Stack
The stack is: 0
The stack is: 1 0
The stack is: 2 1 0
The stack is: 3 2 1 0
3 popped from stack
The stack is: 2 1 0
2 popped from stack
The stack is: 1 0
1 popped from stack
The stack is: 0
0 popped from stack
The stack is: empty

```

图 22.10 堆栈实现 - fig22_10.py

22.5 队列

“队列”类似于在超市出口排队结账的人——队首的人先结账，其他客户在队尾加入队列，并等候结账。队列节点只能从队首移除，在队尾插入，这称为“先入先出”（FIFO）数据结构。插入和移除操作分别叫做“入队”（enqueue）和“出队”（dequeue）。

队列在计算机系统中有许多应用。大多数计算机都只配备一个处理器，所以每次只能为一个用户服务。新进入的用户放在队尾，逐渐向前移动，并等候服务。位于队首的总是下一个要获得服务的用户。

队列也用于支持脱机打印。多用户环境可能只配备一台计算机，但许多用户都可同时生成打印输出。如打印机忙，其他输出仍可正常生成，只是“脱机打印”到磁盘，并在一个队列中等待打印机可用。

在计算机网络中，数据包也可在队列中等待。每次有一个数据包抵达网络节点时，都必须路由传递到下一个节点，直到抵达终点。假定路由器每次只能传递一个数据包，那么其他数据包必须入队等待，直到路由器可用。

计算机网络上的文件服务器可同时处理来自多个客户的文件访问请求。但为客户请求提供服务时，服务器的能力是有限的。一旦超过这个能力，新的客户请求必须入队等待。

图 22.11 主要通过“合成”来创建 Queue 类。该类有一个私有数据成员，即 `_queueList`，它是 List 类（在图 22.3 定义）的一个对象。我们希望 Queue 提供 enqueue（入队）和 dequeue（出队）方法。与此同时，我们注意到这些方法本质上就是 List 类的 `insertAtBack` 和 `removeFromFront` 方法。所以在实现 Queue 的方法时，我们让每个方法都调用 `_queueList` 对象的合适方法；也就是说，让 enqueue 调用 `insertAtBack`，让 dequeue 调用 `removeFromFront`。当然，List 类还包含其他方法（即 `insertAtFront` 和 `removeFromBack`），它们是 Queue 类不会调用的。driver 程序（图 22.12）使用 Queue 类实例化一个队列对象。程序按先入先出顺序，使 0~3 的整数先入队，再出队。

```

1 # Fig. 22.11: QueueModule.py
2 # Class Queue definition.
3
4 from ListModule import List
5
6 class Queue:
7     """Queue composed from linked list"""
8
9     def __init__( self ):
10         """Queue constructor"""
11
12         self._queueList = List()

```

```

13
14     def __str__( self ):
15         """Queue string representation"""
16
17         return str( self._queueList )
18
19     def enqueue( self, element ):
20         """Enqueue element"""
21
22         self._queueList.insertAtBack( element )
23
24     def dequeue( self ):
25         """Dequeue element"""
26
27         return self._queueList.removeFromFront()
28
29     def isEmpty( self ):
30         """Return 1 if Queue is empty"""
31
32         return self._queueList.isEmpty()

```

图 22.11 队列实现 - QueueModule.py

```

1 # Fig. 22.12: fig22_12.py
2 # Driver to test class QueueModule.
3
4 from QueueModule import Queue
5
6 queue = Queue()
7
8 print "processing a Queue"
9
10 for i in range( 4 ):
11     queue.enqueue( i )
12     print "The queue is:", queue
13
14 while not queue.isEmpty():
15     dequeue = queue.dequeue()
16     print dequeue, "dequeued"
17     print "The queue is:", queue

```

```

processing a Queue
The queue is: 0
The queue is: 0 1
The queue is: 0 1 2
The queue is: 0 1 2 3
0 dequeued
The queue is: 1 2 3
1 dequeued
The queue is: 2 3
2 dequeued
The queue is: 3
3 dequeued
The queue is: empty

```

图 22.12 队列实现 - fig22_12.py

22.6 树

链表、堆栈和队列都是线性数据结构。“树”(Tree)则是非线性的二维数据结构，具有一些特殊属性。树节点包含两个或更多的链接。本节讨论了“二叉树”(图 22.13)，其中所有节点都包含两个链接(可能有一个为 None，也可能全部为 None)。“根节点”是树的第一个节点。根节点中的每个链接都引用一个“子”。“左子”是“左子树”的根节点，而“右子”是“右子树”的根节点。同一个节点的子节点称为“同辈节点”。没有子节点的节点称为“叶节点”。但描绘树结构时，往往从根节点开始向下描绘；刚

好和自然界的树相反。

本节要创建一种特殊二叉树，名为“二叉搜索树”(BST)。二叉搜索树(无重复节点值)的特点是，任何左子树中的值都要小于子树的父节点的值，任何右子树中的值都要大于子树的父节点的值。图 22.14 展示了含有 12 个值的一个二叉搜索树。注意二叉搜索树的形状是不定的，具体取决于值插入树中的顺序。

常见编程错误 22.3 不将叶节点的链接设为 None 是逻辑错误。

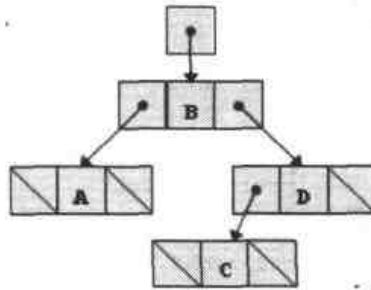


图 22.13 二叉树示意图

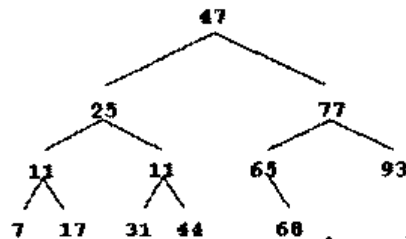


图 22.14 二叉搜索树示意图

图 22.15 将 Tree 类定义成二叉搜索树。图 22.16 的程序创建 Tree 类的一个对象，并采取 3 种方式遍历——中序 (Inorder)、前序 (Preorder) 和后序 (Postorder)。

```

1 # Fig. 22.15: TreeModule.py
2 # Treenode and Tree definition.
3
4 class Treenode:
5     """Single Node in a Tree"""
6
7     def __init__( self, data ):
8         """Treenode constructor"""
9
10        self._left = None
11        self._data = data
12        self._right = None
13
14    def __str__( self ):
15        """Tree string representation"""
16
17        return str( self._data )
18
19 class Tree:
20     """Binary search tree"""
21
22    def __init__( self ):
23        """Tree Constructor"""
24
25        self._rootNode = None
26
27    def insertNode( self, value ):
28        """Insert node into tree"""
29
30        if self._rootNode is None: # tree is empty

```

```

31     self._rootNode = Treenode( value )
32     else: # tree is not empty
33         self.insertNodeHelper( self._rootNode, value )
34
35     def insertNodeHelper( self, node, value ):
36         """Recursive helper method"""
37
38         if value < node._data: # insert to left
39
40             if node._left is None:
41                 node._left = Treenode( value )
42             else:
43                 self.insertNodeHelper ( node._left, value )
44
45         elif value > node._data:
46
47             if node._right is None: # insert to right
48                 node._right = Treenode( value )
49             else:
50                 self.insertNodeHelper( node._right, value )
51
52         else: # duplicate node
53             print value, "duplicate"
54
55     def preOrderTraversal( self ):
56         """Preorder traversal"""
57
58         self.preOrderHelper( self._rootNode )
59
60     def preOrderHelper( self, node ):
61         """Preorder traversal helper function"""
62
63         if node is not None:
64             print node,
65             self.preOrderHelper( node._left )
66             self.preOrderHelper( node._right )
67
68     def inOrderTraversal( self ):
69         """Inorder traversal"""
70
71         self.inOrderHelper( self._rootNode )
72
73     def inOrderHelper( self, node ):
74         """Inorder traversal helper function"""
75
76         if node is not None:
77             self.inOrderHelper( node._left )
78             print node,
79             self.inOrderHelper( node._right )
80
81     def postOrderTraversal( self ):
82         """Postorder traversal"""
83
84         self.postOrderHelper( self._rootNode )
85
86     def postOrderHelper( self, node ):
87         """Postorder traversal helper function"""
88
89         if node is not None:
90             self.postOrderHelper( node._left )
91             self.postOrderHelper( node._right )
92             print node,

```

图 22.15 二叉树实现 - TreeModule.py

```

1 # Fig. 22.16: fig22_16.py
2 # Driver to test Tree class.
3
4 from TreeModule import Tree
5

```



```

6 tree = Tree()
7 values = raw_input(
8     "Enter 10 integer values, separated by spaces:\n" )
9
10 for i in values.split():
11     tree.insertNode( int( i ) )
12
13 print "\nPreorder Traversal"
14 tree.preOrderTraversal()
15 print
16
17 print "Inorder Traversal"
18 tree.inOrderTraversal()
19 print
20
21 print "Postorder Traversal"
22 tree.postOrderTraversal()
23 print

```

```

Enter 10 integer values, separated by spaces:
50 25 75 12 33 67 88 6 13 68

Preorder Traversal
50 25 12 6 13 33 75 67 68 88
Inorder Traversal
6 12 13 25 33 50 67 68 75 88
Postorder Traversal
6 13 12 33 25 68 67 88 75 50

```

图 22.16 二叉树实现 - fig22_16.py

图 22.16 中的主程序首先实例化名为 `tree` 的一个二叉树（第 6 行）。程序提示输入 10 个整数，每个整数都通过调用 `insertNode` 插入二叉树（第 10~11 行）。第 23~24 行对 `tree` 执行前序、中序和后序遍历（稍后会详加解释）。

来看看类定义（图 22.15）。`TreeNode` 类的数据成员是节点值（`_data` 属性）；另外还有两个引用成员：`_left` 引用节点的左子树，`_right` 引用节点的右子树。构造函数（第 7~12 行）将 `_data` 成员设为作为构造函数的参数提供的值，将 `_left` 和 `_right` 引用设为 `None`，这样可将该节点初始化为叶节点。

`Tree` 类有数据成员 `_rootNode`，它引用树的根节点。`Tree` 构造函数（第 22~25 行）将 `_rootNode` 初始化成 `None`，表明树最开始是空的。`Tree` 类有 `insertNode` 方法（第 22~25 行），用于在树中插入一个新值。另外还有 `preorderTraversal`（第 55~58 行）、`inorderTraversal`（第 68~71 行）以及 `postorderTraversal`（第 81~84 行）方法，它们分别按指定的顺序遍历树。每个方法都要调用自己的、独立的递归工具方法，以便针对树的内部表示执行恰当的操作。

注意，节点只能作为叶节点插入二叉搜索树。`insertNode` 方法在树中插入一个节点。如果树是空的，方法就新建一个 `TreeNode`，并把它插入树（第 31 行）。否则，方法就调用工具方法 `insertNodeHelper`（第 35~53 行），在树中递归地插入值。

如果树不是空的，程序将要插入的值和根节点的值比较。如插入值较小，程序递归调用 `insertNodeHelper`，将值插入左子树（第 43 行）。如插入值较大，程序递归调用 `insertNodeHelper`，将值插入右子树（第 50 行）。如果要插入的值等于根节点的值，程序打印“duplicate”（重复）消息，并直接返回，不将重复的值插入树（第 52~53 行）。

`inOrderTraversal`、`preOrderTraversal` 和 `postOrderTraversal` 等方法都遍历树并打印节点值。

`inOrderTraversal` 的步骤如下：

1. 用 `inOrderTraversal` 遍历左子树。
2. 处理节点值（即打印它）。
3. 用 `inOrderTraversal` 遍历右子树。

只有处理完左子树的值，才会处理节点值。对于图 22.17 的树，`inOrderTraversal` 结果是：

6 13 17 27 33 42 48

注意二叉搜索树的 `inOrderTraversal` 按升序打印节点值。创建二叉搜索树的过程实际就是数据排序的过程——所以也把它称为“二叉树排序”。

`preOrderTraversal` 的步骤如下：

1. 处理节点值。
2. 用 `preOrderTraversal` 遍历左子树。
3. 用 `preOrderTraversal` 遍历右子树。

访问每个节点时，都要对它的值进行处理。一个节点的值处理完后，先处理它的左子树的值，再处理它的右子树的值。对于图 22.17 的树，`preOrderTraversal` 结果是：

27 13 6 17 42 33 48

`postOrderTraversal` 的步骤如下：

1. 用 `postOrderTraversal` 遍历左子树。
2. 用 `postOrderTraversal` 遍历右子树。
3. 处理节点值。

每个节点值只有在其所有子节点值打印后才会打印。对于如图 22.17 所示的树，`postOrderTraversal` 的结果是：

6 17 13 33 48 42 27

二叉搜索树简化了重复值的消除。树在创建时就能检测到插入重复值的试图，因为每次和根节点值比较时，都要根据是较小还是较大，而决定分别进入左边还是右边。如果与根节点的值相等，就表明是重复的。

可在二叉树中快速地搜索与一个键值匹配的值。如果树是平衡的，每一级都包含约两倍于上一级的值。所以对于 n 个元素的二叉搜索树，它最多只有 $\log_2 n$ 级；换言之，最多进行 $\log_2 n$ 次比较，就可找到一个匹配项，或判断出没有任何匹配。例如搜索 1000 个元素的平衡二叉搜索树时，最多只需执行 10 次比较，因为 $2^{10} > 1000$ 。即使元素数量提升到 1000000，最多也只需执行 20 次比较，因为 $2^{20} > 1000000$ 。

另外，还可对二叉树执行“级序遍历”，即从根节点一级开始，逐级访问二叉树节点。在树的每一级，都按从左到右的顺序访问每个节点。

第 23 章 案例分析：网上书店

学习目标

- 使用 Python 和 CGI，建立一个三层的、客户/服务器结构的、分布式的 Web 应用程序
- 理解 HTTP 会话概念
- 会用 Session 类在不同页之间跟踪 HTTP 会话
- 使用脚本创建 XML，并用 XSLT 将 XML 转换成可由客户显示的格式
- 在 Apache Web 服务器上部署应用程序
- 扩展应用程序，使其支持无线客户端

23.1 概述

本章要实现一个书店 Web 应用程序，它综合运用了以前讲解的大量技术，标志着本书 Python 学习之旅的最高成就。运用的技术包括 CGI(第 6 章)、XML、XSL 和 XSLT(第 15~16 章)、MySQL 和 Python DB-API(第 17 章)、XHTML 和 CSS。这个案例分析还展示了另外一些特性，比如 WML(无线标记语言)和 XHTML Basic。新元素会在需要用到时当场讲解。该应用程序部署在 Apache Web 服务器上。完成本章的学习后，马上就可以在 Apache Web 服务器上实现一个实用的 Web 应用程序。

23.2 HTTP 会话和会话跟踪技术

网站要想提供电子商务应用，需要有能力生成自定义网页，并提供满足客户需要的功能。这种网站应用程序的一个例子就是我们为本章的网上书店案例分析而设计的购物车。这个应用程序为了正确处理订单，要求服务器必须区分不同的客户，确保发货和付款正确进行。利用会话跟踪技术，服务器可区分不同的客户。本节介绍了两种会话跟踪方法，即“Cookie”和“嵌入状态信息”。另外，还解释了它们如何通过 Internet 协议工作。

Web 浏览器向 Web 服务器发出的每个请求都创建一个新连接。服务器处理完客户请求后，连接中断。如 Web 服务器需要在几个不同的请求之间维持客户信息，客户每次都要表明自己的身份。

表明身份的一个办法是使用 Cookie，这是一种非常小的文本文件，由服务器(或网站)作为请求响应的一部分发送。客户首次访问网站时，网站可在客户的计算机上存储一个 Cookie，以记录用户首选项或其他信息(例如客户的用户名)。以后访问网站时，就可直接提取 Cookie 中的信息。例如，许多网站都用 Cookie 存储客户的邮政编码。以后访问网站时，就可根据邮编提供用户所在地区的天气和新闻信息。

客户和服务器之间每个基于 HTTP 的交互都包括一个“头”(Header)，其中含有与请求有关的信息(从客户到服务器的通信)，或者与响应有关的信息(从服务器到客户的通信)。Python 脚本收到请求时，头内含有的信息包括请求类型(get 或 post)，以及服务器以前存储在客户计算机上的 Cookie。服务器生成响应时，头信息则含有服务器希望存储到客户计算机上的任何 Cookie。

取决于 Cookie 的有效期限，Web 浏览器要么在浏览会话期间(也就是在用户关闭 Web 浏览器之前)保持 Cookie，要么将 Cookie 存储到客户计算机上以便将来使用。浏览器向服务器发出一个请求时，要向服务器返回在以前的交互过程中，由服务器存储到客户机上的任何 Cookie。一旦 Cookie 过期(即超过它的截止期限)，Web 浏览器就把它删除。

Cookie 是 Web 应用程序对客户进行区分的最容易的方式。然而，并不是所有客户计算机或 Web 浏览器都支持 Cookie。例如，有的用户关心自己的隐私和安全，所以在 Web 浏览器中禁用了 Cookie。所以，如果网站只用 Cookie 同客户交互，这一部分用户就无法正常访问网站。正是因为考虑到这一点，我

们决定在网上书店应用中不使用 Cookie 来跟踪会话。

移植性提示 23.1 有的浏览器（尤其是版本较老的）不支持 Cookie。如果服务器只依赖 Cookie，部分用户会被拒之门外。

另一种会话跟踪方法是在客户和服务器的每次通信中嵌入“状态信息”。状态信息可能包括用户名、密码或者能在用户返回网站时提供帮助的其他信息。客户首次连接服务器时，Web 应用程序为客户指派一个不重复的会话 ID。发出更多请求时，客户的会话 ID 会和存储在服务器内存中的会话 ID 比较。

ID 必须从一个页传到下一页，这样 Web 应用程序才能跟踪当前客户的会话 ID，由此对不同客户进行区分。可用不同的方式来做这一点。传递 ID 的一种方式是把它放在每个网页的一个隐藏表单字段中。Web 应用程序可将 ID 视为普通 CGI 参数访问。另一种方式是在指向下一个页的超链接中添加这个 ID。这样，下一个页就可从 URL 中提取出 ID。要了解 CGI 参数传递的详情，请参见第 6 章。

用会话 ID 跟踪会话信息既有优点，也有缺点。优点是 Web 浏览器无法禁用会话 ID。缺点是含有会话 ID 的网页地址较长，而且每个超链接都要嵌入这种 ID。一些较老的浏览器不支持过长的 URL，所以可能会出问题。另一个缺点是嵌入这种信息会带来潜在的安全风险。在网页或 URL 中存储会话 ID，任何人都可以看到这个 ID，并访问用户的数据。但是，尽管存在这些缺点，我们的网上书店仍然采用这种方法来跟踪 HTTP 会话。

良好编程习惯 23.1 Cookie 和嵌入状态信息各有优缺点。决定之前，要仔细考察每一种技术。

23.3 在网上书店中跟踪会话

构建 Web 应用程序之前，先讨论一下 Session 类，它利用嵌入状态信息的技术来跟踪 HTTP 会话（图 23.1）。用户首次进入网上书店，会新建一个 Session 对象以建立会话跟踪。新的 Session 对象包含不重复的会话 ID，以及由会话数据构成一个字典。会话数据序列化并存储到服务器上。有关序列化更多的信息，请参见第 14 章。客户在书店中浏览时，会话 ID 会作为查询字符串的一部分，在不同脚本之间传递。后续每个脚本都可提取 ID，并获得会话信息。马上就要讲到，这是用参数 1 创建一个 Session 对象来实现的。

Session 实例创建时，如果 createNew（传给构造函数的参数）包含的值不是 0（默认值），就新建一个会话。在这种情况下，首先要调用 generateID 方法（第 82 行）。generateID 方法（第 146~153 行）使用 sha 模块生成不重复的 ID 值。第 150~151 行创建一个字符串，其中包括会话时间、客户地址以及客户端口。然后，第 152~153 行使用该字符串创建并返回一个不重复的 ID。客户以后同应用程序交互时，Web 应用程序将根据这个 ID 来识别客户。第 21 章展示了 sha 的应用。

```

1 # Fig. 23.1: Session.py
2 # Contains a Session class that keeps track of an http session
3 # by assigning a session ID and pickling session information.
4
5 import os
6 import sha
7 import cgi
8 import time
9 import urlparse
10 import urllib
11 import cPickle
12
13 def getClientType():
14     """Return client type and corresponding file extension"""
15
16     # search environment variables for identifying strings
17     if os.environ["HTTP_USER_AGENT"].find("MSIE") > -1 or \
18         os.environ["HTTP_USER_AGENT"].find("Netscape") > -1:
19
20         # MSIE and Netscape represent XHTML clients (.html)

```

```

21     return ( "xhtml", "html" )
22
23     elif os.environ[ "HTTP_ACCEPT" ].find(
24         "text/vnd.wap.wml" ) > -1:
25
26         # text/vnd.wap.wml represents WML clients (.wml)
27         return ( "wml", "wml" )
28
29     else:
30
31         # otherwise, assume XHTML Basic client (.html)
32         return ( "xhtml_basic", "html" )
33
34 def getContentType():
35     """Return the contents of the client's contentType.txt file"""
36
37     # obtain contentType.txt located in client's subfolder
38     try:
39         file = open( getClientType()[ 0 ] + "/contentType.txt" )
40     except:
41         raise SessionError( "Missing file: contentType.txt" )
42
43     contentType = file.read()
44     file.close()
45     return contentType
46
47 def redirect( URL ):
48     """Redirect the client to a relative URL"""
49
50     # use urljoin to append full path to relative URL
51     print "Location: %s\n" % \
52         urlparse.urljoin( "http://" + os.environ[ "HTTP_HOST" ] +
53             os.environ[ "REQUEST_URI" ], URL )
54
55 class SessionError( Exception ):
56     """User-defined exception for Session class"""
57
58     def __init__( self, error ):
59         """Set error message"""
60
61         # use quote_plus to replace spaces with '+'
62         self.error = urllib.quote_plus( error )
63
64     def __str__( self ):
65         """Return error message"""
66
67         return self.error
68
69 class IDError( Exception ):
70     """User-defined exception for Session class"""
71
72     pass
73
74 class Session:
75     """Session class keeps track of an HTTP session"""
76
77     def __init__( self, createNew = 0 ):
78         """Create a new session or load an existing session"""
79
80         # create new session
81         if createNew:
82             self.sessionID = self.generateID()
83             self.fileName = os.getcwd() + "/sessions/." + \
84                 self.sessionID
85
86         # newly generated ID already exists
87         if self.sessionExists():
88             raise IDError
89
90         self.data = {} # dictionary is empty
91

```

```

92         # store ID, empty cart, content type and agent type
93         self.data[ "ID" ] = self.sessionID
94         self.data[ "cart" ] = {}
95         self.data[ "content type" ] = getContentType()
96         self.data[ "agent" ], self.data[ "extension" ] = \
97             getClientType()
98
99     # attempt to load previously created session
100     else:
101
102         # session ID is passed in query string
103         queryString = cgi.parse_qs( os.environ[ "QUERY_STRING" ] )
104
105         # no ID has been supplied in query string
106         if not queryString.has_key( "ID" ):
107             raise SessionError( "No ID given" )
108
109         self.sessionID = queryString[ "ID" ][ 0 ]
110         self.fileName = os.getcwd() + "/sessions/." + \
111             self.sessionID
112
113         # supplied ID is invalid
114         if not self.sessionExists():
115             raise SessionError( "Nonexistent ID given" )
116
117         # load pickled session dictionary
118         self.loadSession()
119
120     def sessionExists( self ):
121         """Determine if the specified session file exists"""
122
123         return os.path.exists( self.fileName )
124
125     def loadSession( self ):
126         """Unpickle dictionary of existing session"""
127
128         if self.sessionExists():
129             sessionFile = open( self.fileName )
130             data = cPickle.load( sessionFile )
131             sessionFile.close()
132             self.data = data
133
134     def saveSession( self ):
135         """Pickle session dictionary to session file"""
136
137         sessionFile = open( self.fileName, "w" )
138         cPickle.dump( self.data, sessionFile )
139         sessionFile.close()
140
141     def deleteSession( self ):
142         """Delete session file"""
143
144         os.remove( self.fileName )
145
146     def generateID( self ):
147         """Use sha to generate a unique ID"""
148
149         # generate ID using time, client address and port
150         randomString = str( time.time() ) + \
151             os.environ[ "REMOTE_ADDR" ] + os.environ[ "REMOTE_PORT" ]
152         ID = sha.new( randomString )
153         return ID.hexdigest()

```

图 23.1 跟踪 HTTP 会话的工具函数和 Session 类

Session 从 generateID 获得新 ID 后, 就根据 ID 生成会话文件的名称, 即 fileName, 并判断会话是否存在 (第 87 行)。如果会话文件已经存在, Session 引发用户自定义的 IDError 异常 (第 88 行)。注意会话文件名是在句点(.)之后添加会话 ID。所有会话文件都存储到当前工作目录 (即 cgi-bin) 的 sessions

子目录中。每个文件都包括一个序列化的会话字典，用于一个不同的 HTTP 会话。每个会话字典都包含了相关的客户信息，比如客户类型（例如 XHTML）以及表示成字典的用户购物车。在用户使用书店应用程序的过程中，每个脚本都用会话 ID 来获得用户的会话文件。这样，客户信息可由应用程序中的每个脚本使用。

Session 类在 data 字典中存储会话信息。第 90 行通过创建一个空的会话字典来初始化 data。第 93 行将会话 ID 存储到这个字典中。第 94 行创建一个空字典来表示一个空购物车。第 95 行将 getContentType 函数的结果存储到会话字典。getContentType 函数（第 34~45 行）打开 contentType.txt 文件（该文件存储在根据客户类型命名的一个子目录中），并返回该文件的内容。contentType.txt 中包含一个特定于浏览器的 HTTP 标头，它必须附加到发送给客户的每个页之前。图 23.2 是该文件的一个例子。第 96~97 行从 getClientType 函数获得客户类型，并把它存储到会话字典中。getClientType 函数（第 13~32 行）在 HTTP_USER_AGENT 和 HTTP_ACCEPT 环境变量中搜索特定的值，以确定客户类型，并返回一个二元元素组（由客户类型及其对应的文件扩展名构成）。通过存储与客户类型有关的信息，可保证网上书店应用程序中的每个脚本都能生成客户有能力呈现的输出。

```
1 Content-type: text/html
2
```

图 23.2 用于 XHTML 客户的 contentType.txt

为了在不同的页之间保存会话数据，程序必须调用 saveSession 方法（第 134~139 行）。该方法新建一个和 fileName 属性值对应的会话文件。第 138 行使用 cPickle 模块序列化会话字典（self.data），并把它 dump 到会话文件中。用 cPickle 来存储会话信息，可将 Python 对象与会话轻松地保存下来。

应用程序为客户建立一个会话 ID 后，其余脚本在创建 Session 对象时，要将 createNew 设为 0（默认）。这样一来，Session 对象会从相应的会话文件中载入现成的状态信息。在这种情况下，就从第 103 行开始执行。会话 ID 将作为查询字符串的一部分在不同脚本之间传递。第 103 行获得查询字符串并解析它。如果没有指定 ID，构造函数引发一个 SessionError 异常（第 107 行）；否则就提取出会话 ID，并判断文件名（第 109~111 行）。SessionError 提供一个 error 属性，其中包含指定的错误消息。这有助于用户理解出错原因（参见 23.13 节）。然后，第 114 行检查指定的会话 ID 是否存在。如果会话不存在，构造函数引发一个 SessionError 异常（第 115 行）；否则，构造函数就调用 loadSession 方法。该方法（第 125~132 行）打开会话文件（第 129 行）。然后，使用 cPickle 载入包含的会话字典（第 130 行）。该字典保存在会话字典中（self.data）。

如果 Web 应用程序不再需要保存一个特定的会话，可调用 deleteSession 方法（第 141~144 行）来删除那个会话。该方法通过调用 os.remove 来删除会话文件。

我们使用 redirect 函数将客户重定向到另一个页，它可将客户重定向到一个相对 URL（第 47~53 行）。redirect 函数为应用程序中的各个脚本提供了方便的重定向机制。第 52~54 行使用 urlparse.urljoin 函数将基本 URL 与指定的 URL 合并到一起。基本 URL 是从 HTTP_HOST 和 REQUEST_URI 环境变量获得的。然后，通过一个 Location 头来实现重定向。

23.4 网上书店体系结构

本节概述书店应用程序的体系结构。图 23.3 展示了各 Python 脚本之间的交互。另外，图 23.4 总结了本案例分析中使用的各个文件。

这个应用程序包含一系列 XHTML 文档、XHTML Basic 文档、WML 文档、XSLT 文档以及 Python 脚本，这些文档相互协作，建立起出售 Deitel 出版物的网上书店。这是一个分布式的三层 Web 应用程序。用户的 Web 浏览器是“客户层”，也称为“顶层”。浏览器要么显示静态文档，要么显示动态文档，以便与服务器进行交互。应用程序根据客户端类型创建这些文档。“服务器层”也称为“中间层”，由几个代表客户执行操作的脚本程序构成。这些脚本执行的任务包括创建出版物列表、创建包含出版物细节的文

档、在购物车中添加商品、查看购物车以及处理最后的订单等等。“信息层”也称为“数据层”或者“底层”，它负责维护应用程序需要的数据。信息层使用了第 17 章开发的 Books 数据库。

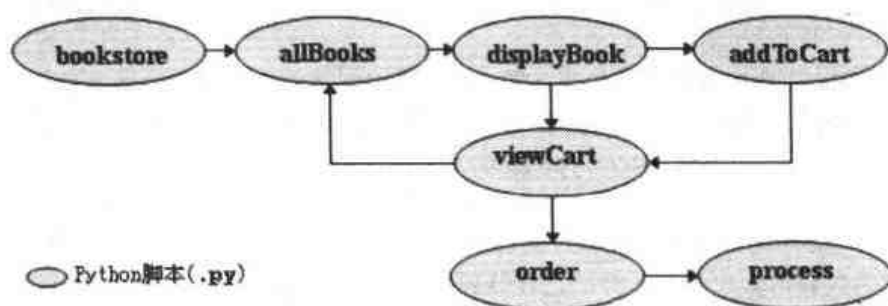


图 23.3 书店应用程序各组件的交互

图 23.3 展示了书店应用程序各个组件之间的交互。应用程序为用户创建一个 Session 后，就将用户重定向至 allBooks.py，该脚本与数据库交互，动态地创建一个书籍列表。结果是含有书籍列表的一个 XML 文档。然后，参照特定于浏览器的 XSLT 样式表（allBooks.xsl）对这个 XML 文档进行处理。生成的页中包含到 displayBook.py 的链接。displayBook.py 接收所选书籍的 ISBN 号，并用 ISBN 提取那本书的信息，为所选的书籍生成另一个 XML 文档。然后，参照另一个特定于浏览器的 XSLT 样式表（displayBook.xsl）对这个 XML 文档进行处理，生成含有书籍相关信息的一个文档。在此文档中，用户可使用 GUI 组件（即按钮）将当前书籍放到购物车中，或者查看购物车。

将书籍添加到购物车，会调用 addToCart.py；查看购物车的内容，会调用 viewCart.py，它返回特定于浏览器的一个文档（同样地，参照 XSLT 对 XML 进行处理而获得），其中列出了购物车的内容、每件商品的小计金额以及所有商品的总金额。在购物车中添加一件商品后，addToCart.py 会处理用户的请求，然后将请求转发给 viewCart.py，后者创建文档，以显示当前购物车内容。之后，用户要么继续购物（allBooks.py），要么去结账（order.py）。如选择结账，会向用户显示一个表单，以便输入姓名、地址和信用卡信息。用户提交这个表单后，会调用 process.py 对其进行处理。该脚本向用户发送一个确认文档以结束交易。图 23.4 总结了这些脚本以及系统中使用的其他文件。

文件	说明
Session.py	包含 Session 类。该类的一个实例为每个用户都指派不重复的会话 ID，并将与每个 ID 对应的数据序列化到一个字典中，从而对 HTTP 会话进行跟踪。该文件还包含 3 个工具函数，用于重定向客户、判断用户的客户端类型以及判断客户端的内容类型（存储在 contentType.txt 中）
contentType.txt	用一行文本指定数据的内容类型。会为每种客户类型都创建这样的一个文件
bookstore.py	这是书店应用程序的默认主页。在此，为用户创建了一个新的 Session，以跟踪 HTTP 会话。然后，将用户重定向至 allBooks.py
styles.css	这是一个层叠样式表（CSS）文件，链接到要在客户端呈现的所有 XHTML 和 XHTML Basic 文档。CSS 文件可在所有静态和动态文档中应用统一的格式
allBooks.py	该脚本使用 Book 实例创建一个包含产品列表的文档。它查询 Books 数据库，获得书籍的列表。结果在处理之后，放到一个 Book 实例列表中。列表作为客户的一个会话属性存储下来。脚本创建一个 XML 文档，它代表所有书籍。然后，为 XML 应用一个特定于浏览器的 XSL 转换（allBooks.xsl），生成可由客户端正确呈现的文档
allBooks.xsl	利用这个 XSLT 样式表，将整个书籍目录的 XML 表示转换成可由客户端浏览器正确呈现的一个文档。针对每种客户类型，都存在这样的一个文件
Book.py	包含 Book 类。该类的一个实例代表用于一本书的数据。Book 的 getXML 方法会返回代表书的一个 XML Element

文件	说明
displayBook.py	该脚本包含用户选择的一本书的 XML 表示，然后为 XML 应用一个特定于浏览器的 XSL 转换（displayBook.xsl），生成可由客户呈现的一个文档
displayBook.xsl	这个 XSLT 样式表将一本书的 XML 表示转换成浏览器可以正确呈现的一个文档。针对每种客户类型，都存在这样的一个文件
CartItem.py	包含 CartItem 类。该类的实例负责维护一个 Book，以及该 Book 当前在购物车中的数量。CartItem 存储在代表购物车内容的一个字典中。
addToCart.py	该脚本负责更新购物车。如购物车中已有代表所选商品的一个 CartItem，脚本只需更新它的数量。否则，脚本就新建一个数量为 1 的 CartItem。更新了购物车之后，用户会被重定向到 viewCart.py，以查看当前购物车内容
viewCart.py	该脚本从购物车提取 CartItem，小计购物车中的每种商品，总计购物车中的所有商品，并创建一个 XML 文档来表示购物车中的所有商品。然后，这个脚本为 XML 应用一个特定于浏览器的 XSL 转换（viewCart.xsl），生成可由客户端呈现的文档
viewCart.xsl	这个 XSLT 样式表将购物车中所有 CartItem 的 XML 表示转换成客户端浏览器可以呈现的一个文档。针对每种客户类型，都存在这样的一个文件
order.py	查看购物车时，用户可单击 Check Out（结账）按钮以执行该脚本。脚本会显示特定于浏览器的订单表单。在本例中，该表单没有实际功能
orderForm.html, orderForm.wml	这些静态文档包含了订单表单，它们是由 order.py 显示的
process.py	这个脚本模拟处理用户的信用卡信息，并载入一个特定于浏览器的文档，指出订单已被处理，并显示总计订购金额
thankYou.html, thankYou.wml	这些静态文档由 process.py 显示，显示订单已被处理，并指出总计订购金额
error.py	发生错误时执行。它创建用于表示错误的一个 XML 文档。然后参照一个特定于浏览器的 XSLT 样式表（error.xsl）对 XML 进行处理，以生成可由客户呈现的一个文档。该文档向用户指出发生了一个错误
error.xsl	这个 XSLT 样式表将指出错误的 XML 文档转换成可由客户浏览器呈现的一个文档。针对每种客户类型，都存在这样的一个文件

图 23.4 书店系统的各个组件

23.5 配置网上书店

本书包括网上书店在内的所有示例程序都可从 Deitel 网站下载，网址为 www.deitel.com/books/downloads.html。在与第 23 章对应的示例目录中，可找到书店系统用到的所有文件。要在 Apache 上设置书店，请将htdocs子目录的内容拷贝到 Apache 的 HTML 文档目录的根。在 Windows 平台上，这个根目录是htdocs，通常位于 C:\Program Files\Apache Group\Apache 中；在 Linux 平台上，这个根目录可能叫做htdocs或者html，并可能位于两个目录中：/var/www 或者/usr/local/httpd。注意，书店系统的大多数文件要依赖于这些目录的名称和相对位置。如系统目录结构和代码中指定的不同，可能需要对源代码进行一些修改。

接着，将cgi-bin子目录的内容从与第23章对应的示例目录拷贝到 Apache 的cgi-bin目录。在 Windows 平台上，cgi-bin 目录一般在 C:\Program Files\Apache Group\Apache 中；在 Linux 上，cgi-bin 可能在两个目录中：/var/www 或者/usr/local/httpd。拷贝完成后，请启动 Apache。图 23.5 展示了 Apache 在一个 Windows 平台上的目录结构。

注意，为运行书店应用程序，需要用到以下软件：4Suite（第 16 章）、PyXML（第 16 章）、MySQL（第 17 章）以及 Books 数据库（第 17 章）。部署书店应用程序之前，要先安装好这些软件；要获得具

体的软件安装指南, 请访问 www.deitel.com。

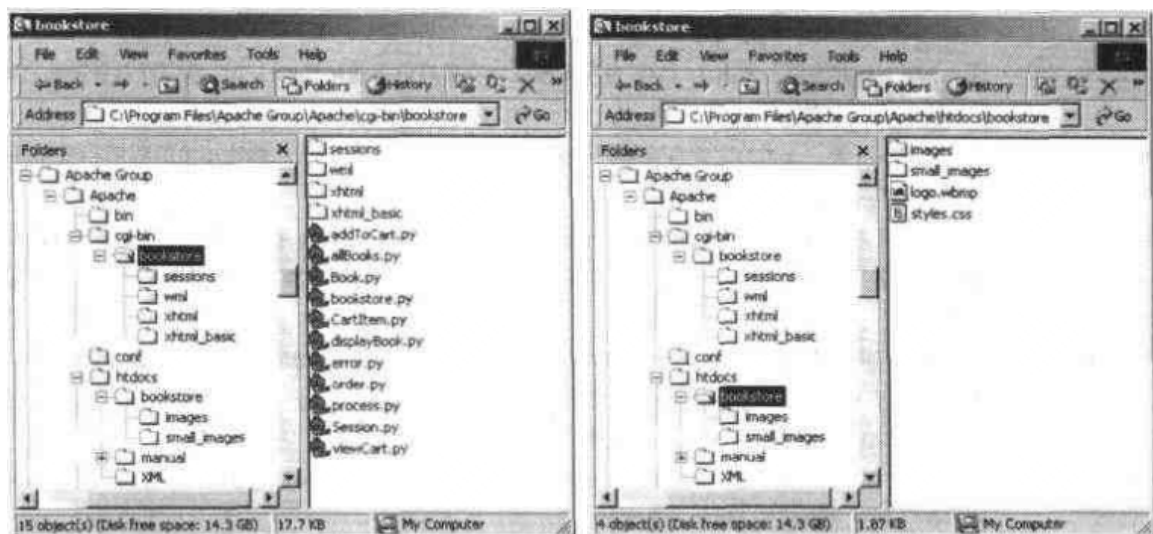


图 23.5 用于网上书店的 Apache 目录结构

23.6 进入网上书店

图 23.6 (bookstore.py) 是书店的默认主页, 即所谓的“欢迎文件”, 因为它是用户访问书店时看到的第一个文件。在 Apache 上安装好书店应用程序后, 请在 Web 浏览器中输入以下 URL 来访问主页:

<http://localhost/cgi-bin/bookstore/bookstore.py>

第 9 行将 session 初始化成 None, 表明此时尚未新建一个 Session。for 结构尝试为客户创建一个独一无二的会话 (第 12~28 行)。第 16 行新建一个 Session。要记住, 如会话生成的 ID 已存在, 会引发一个 IDError 异常 (参见 23.3 节)。在这种情况下, 程序将休眠 0.1 秒 (使 sha 使用的字符串发生改变), 并再次尝试。如连续三次尝试失败, 第 36 行将客户重定向到 error.py, 该脚本向客户显示一条错误消息 (参见 23.13 节)。如发生了 SessionError 异常 (例如 contentType.txt 不存在), 第 24 行将定向重定向到 error.py。

如果没有发生任何异常, 程序退出循环 (第 28 行), 并执行第 32~34 行。第 32 行创建重定向字符串, 以便将客户重定向到 allBooks.py。会话 ID 作为查询字符串的一部分存储在 URL 中, 从而确保 allBooks.py 能识别用户的身份。第 33~34 行保存会话, 并调用 Session 的 redirect 函数, 将客户重定向到 allBooks.py, 后者列出可选书籍。

```

1  #!c:\Python\python.exe
2  # Fig. 23.6: bookstore.py
3  # Create a new Session for client.
4
5  import sys
6  import time
7  import Session
8
9  session = None # Session not yet created
10
11 # attempt to create Session three times
12 for i in range( 3 ):
13
14     # create new Session
15     try:
16         session = Session.Session( 1 )
17

```

```

18     # ID already exists
19     except Session.IDError:
20         time.sleep( 0.1 ) # wait 0.1 seconds -- try again
21
22     # missing content type
23     except Session.SessionError, message:
24         Session.redirect( "error.py?message=%s" % message )
25         sys.exit()
26
27     else:
28         break # Session created successfully
29
30 # if successful, save Session and re-direct to allBooks.py
31 if session:
32     nextPage = "allBooks.py?ID=%s" % session.data[ "ID" ]
33     session.saveSession()
34     Session.redirect( nextPage )
35 else:
36     Session.redirect( "error.py?message=Unable+to+create+Session" )

```

图 23.6 书店主页 (bookstore.py)

23.7 从数据库获得书籍列表

Python 脚本 allBooks.py 创建一个文档, 其中列出了所有可选书籍。但在获得书籍列表之前, 首先必须为单独一本书创建相应的表示 (图 23.7)。Book 类的一个实例代表特定书籍的属性 (信息), 包括书的 ISBN、标题、版权年、封面图像文件名、版次、出版商 ID 号和定价——虽然本例并没有使用其中的部分信息。Book 的 getXML 方法返回代表书籍的一个 XML Element。

```

1  # Fig. 23.7: Book.py
2  # Represents one book.
3
4  class Book:
5      """A Book instance contains the data for one book"""
6
7      def __init__( self ):
8          """Initialize Book data"""
9
10         self.isbn = None
11         self.title = None
12         self.price = None
13         self.imageFile = None
14         self.copyright = None
15         self.publisherID = None
16         self.editionNumber = None
17
18     def getXML( self, document ):
19         """Return an XML representation of the product"""
20
21         # create dictionary of Book information
22         data = { "isbn" : self.isbn,
23                 "title" : self.title,
24                 "price" : self.price,
25                 "imageFile" : self.imageFile,
26                 "copyright" : self.copyright,
27                 "publisherID" : self.publisherID,
28                 "editionNumber" : self.editionNumber }
29
30         # create product node
31         product = document.createElement( "product" )
32
33         # add element for each Book attribute
34         for key in data.keys():
35
36             # create element, append as child of product
37             temp = document.createElement( key )

```

```

38         temp.appendChild( document.createTextNode(
39             str( data[ key ] ) ) )
40         product.appendChild( temp )
41
42     return product

```

图 23.7 代表书籍信息并定义 XML 表示的 Book 类

getXML 方法（第 18~42 行）使用 DOM 的 Document 和 Element 接口来创建书籍数据的一个 XML 表示。这些数据是 Document 的一部分，以一个参数的形式传递。书的完整信息放在一个 product 元素中（在第 31 行创建）。与书的单项属性对应的元素以“子”的形式追加到 product 元素。

第 34~40 行的 for 结构使用 Document 的 createElement 方法为书的每种属性都创建一个 Element 节点（第 37 行）。第 38~39 行使用 Document 的 createTextNode 方法指定 Element 节点中的文本，并用 Element 的 appendChild 方法将文本追加到 Element 节点。注意，首先使用 str 将数据转换成一个字符串，因为 createTextNode 方法只接受字符串参数。第 40 行调用 Element 的 appendChild 方法，将 Element 节点添加到 product 上。第 42 行向调用者返回 Element 节点 product。图 23.8 显示了由 getXML 方法返回的一个 product 元素的例子。要学习 XML 的更多知识，请参见第 15 章和第 16 章。

创建一个客户会话后，bookstore.py 会将用户重定向到 allBooks.py，后者从 Books 数据库获取书籍列表，并动态生成 XML 文档以表示这些书籍。然后，allBooks.py 脚本参照特定于浏览器的 XSLT 样式表（allBooks.xsl）处理该文档。这样获得的结果就可在客户端正确呈现。图 23.9 展示了 allBooks.py。

```

1 <product>
2   <isbn>0130895601</isbn>
3   <title>Advanced Java 2 Platform How to Program</title>
4   <price>69.95</price>
5   <imageFile>advjhttp1.jpg</imageFile>
6   <copyright>2002</copyright>
7   <publisherID>1</publisherID>
8   <editionNumber>1</editionNumber>
9 </product>

```

图 23.8 product 元素

```

1 #!c:\Python\python.exe
2 # Fig. 23.9: allBooks.py
3 # Retrieve all books from database and store in session.
4 # Display book list to client by retrieving XML and converting
5 # to required format using browser-specific XSLT stylesheet.
6
7 import sys
8 import Book
9 import urllib
10 import Session
11 import MySQLdb
12 from xml.xslt import Processor
13 from xml.dom.DOMImplementation import implementation
14
15 # Load Session
16 try:
17     session = Session.Session()
18 except Session.SessionError, message: # invalid/no session ID
19     Session.redirect( "error.py?message=%s" % message )
20     sys.exit()
21
22 # setup MySQL statement
23 query = """SELECT ISBN, Title, EditionNumber,
24     Copyright, PublisherID, ImageFile, Price
25     FROM Titles ORDER BY Title"""
26
27 # attempt database connection and retrieve list of Books
28 try:
29
30     # connect to the database, retrieve a cursor and execute query
31     connection = MySQLdb.connect( db = "Books" )

```

```

32     cursor = connection.cursor()
33     cursor.execute( query )
34
35     # acquire results and close database connection
36     results = cursor.fetchall()
37     cursor.close()
38     connection.close()
39
40 # error occurred, redirect to error page
41 except MySQLdb.OperationalError, message:
42
43     # replace spaces with '+' for URL compatibility
44     message = urllib.quote_plus( str( message ) )
45     Session.redirect( "error.py?message=%s" % message )
46     sys.exit()
47
48 allBooks = []
49
50 # get row data
51 for row in results:
52
53     # create new Book and set attributes
54     book = Book.Book()
55     book.isbn = row[ 0 ]
56     book.title = row[ 1 ]
57     book.editionNumber = row[ 2 ]
58     book.copyright = row[ 3 ]
59     book.publisherID = row[ 4 ]
60     book.imageFile = row[ 5 ]
61     book.price = row[ 6 ]
62
63     allBooks.append( book ) # one more Book created
64
65 session.data[ "titles" ] = allBooks
66
67 # generate XML
68 document = implementation.createDocument( None, None, None )
69 catalog = document.createElement( "catalog" )
70 document.appendChild( catalog )
71
72 # add all products to catalog
73 for book in allBooks:
74     catalog.appendChild( book.getXML( document ) )
75
76 # process XML against XSLT stylesheet
77 processor = Processor.Processor()
78 style = open( session.data[ "agent" ] + "/allBooks.xsl" )
79 processor.appendStylesheetString( style.read() % \
80     session.data[ "ID" ] )
81 results = processor.runNode( document )
82 style.close()
83
84 # save Session data and display processed XML
85 pageData = session.data[ "content type" ] + results
86 session.saveSession()
87 print pageData

```

图 23.9 allBooks.py 向客户返回包含书籍列表的一个文档

第 16~20 行载入会话。如果会话 ID 未在查询字符串中指定，或者指定的 ID 无效，error.py 会显示一条错误消息（第 19 行）。第 23~25 行准备 MySQL 语句，以便 allBooks.py 查询 Books 数据库时使用。接着，第 31~38 行连接数据库，并获取书籍列表。如果出错（MySQLdb.OperationalError），error.py 会显示一条错误消息，随之程序退出（第 45~46 行）。注意 MySQLdb.OperationalError 返回的错误首先必须转换成一个字符串，其中所有空格替换成+字符。这是因为 MySQLdb.OperationalError 返回一个元组，空格必须替换，使消息在查询字符串中正确传输。

第 51~61 为数据库中的每本书创建一个 Book 实例，并把它追加到 allBooks 列表。第 65 行将 Book

实例列表存储到会话字典 (data)，并将键设为 titles。

接着创建一个 XML 文档，它代表整个书籍目录。xml.dom.DOMImplementation.implementation 的 createDocument 方法创建一个空白 DOM 文档，名为 document (第 68 行)。文档方法 createElement 创建空白目录元素 (第 69 行)。第 70 行将 catalog 元素追加到 document。第 73~74 行获取每本书的 product 元素，并使用 appendChild 方法将元素追加到 catalog。

一个特定于浏览器的 XSLT 样式表处理 XML 文档 (第 77~82 行)。第 77 行创建一个 XSLT 处理器。然后，脚本获取名为 allBooks.xsl 的 XSLT 样式表 (第 78 行)。注意 allBooks.xsl 存储在和客户类型对应的目录中。这可保证 XSLT 样式表将 XML 文档转换成与客户类型兼容的一种格式。第 79~80 行将样式表追加到可接受的样式表列表，以便由 Processor (处理器) 使用。程序必须在样式表中插入会话 ID，因为由样式表转换的 XML 文档不包含会话 ID。将会话 ID 插入样式表后，会话 ID 将在每个超链接中出现。这就保证会话 ID 不至于丢失。第 81 行针对 document 运行 Processor，第 82 行关闭样式表文件。接着，我们向客户显示转换好的 XML。第 85 行创建一个字符串，其中包含 content-type 信息以及处理器结果。第 86~87 行保存会话，并向用户显示页。

图 23.10 列出将 XML 目录表示转换成 XHTML 的 XSLT 样式表，并展示最终生成的 XHTML 文档。

```

1  <?xml version = "1.0"?>
2
3  <!-- Fig. 23.10: allBooks.xsl -->
4  <!-- XSLT style sheet that transforms XML generated by -->
5  <!-- books.py into XHTML. -->
6
7  <xsl:stylesheet version = "1.0"
8    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10   <xsl:output method = "xml" omit-xml-declaration = "no"
11     indent = "yes" doctype-system = "DTD/xhtml1-strict.dtd"
12     doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
13
14   <!-- template for catalog element -->
15   <xsl:template match = "catalog">
16
17     <html xmlns = "http://www.w3.org/1999/xhtml">
18
19       <head>
20         <title>Book list</title>
21         <link rel = "stylesheet" href = "/bookstore/styles.css"
22           type = "text/css" />
23       </head>
24
25       <body>
26
27         <p class = "bigFont">Available Books</p>
28         <p class = "bold">
29           Click a link to view book information</p>
30
31         <!-- match product elements to product template -->
32         <xsl:apply-templates select = "/catalog/product">
33
34           <!-- sort products by title -->
35           <xsl:sort select = "title" />
36
37         </xsl:apply-templates>
38
39       </body>
40
41     </html>
42   </xsl:template>
43
44   <!-- template for building row of product information -->
45   <xsl:template match = "product">
46
47     <a href = "displayBook.py?ID=%s&isbn={isbn}">

```

```

49     <strong><xsl:value-of select = "title" />, <xsl:value-of
50     select = "editionNumber" /></strong>
51   </a><br />
52
53 </xsl:template>
54
55 </xsl:stylesheet>

```

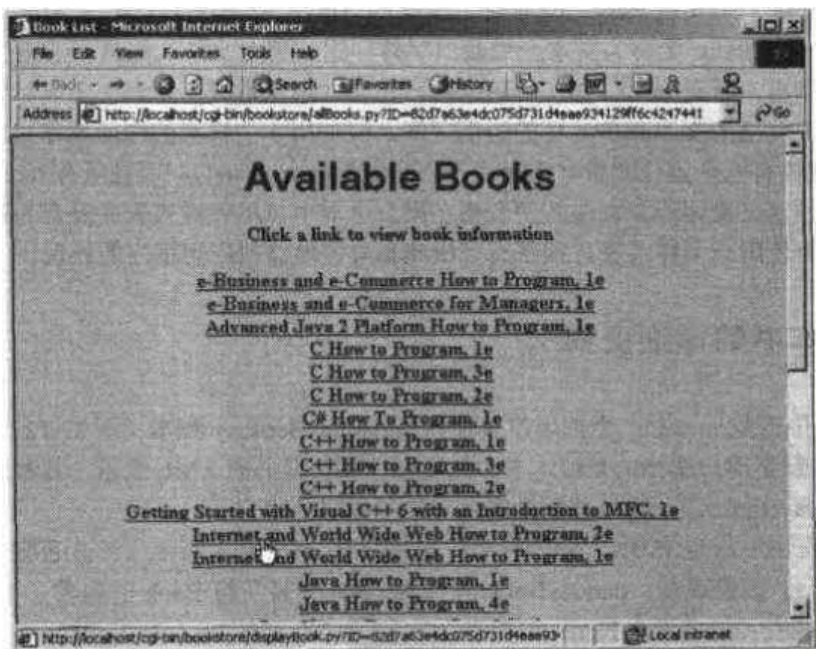


图 23.10 allBooks.xsl 将 XML 格式的目录转换成 XHTML 格式

xsl:template 为 catalog 元素定义了一个模板（第 15~43 行）。在这个模板中，我们插入 product 模板的匹配项（第 32~37 行）。这些匹配项按照它们的 title 元素进行排序（第 35 行），从而确保书籍列表按标题以字母顺序排列。

第 46~53 行为 product 元素定义一个 xsl:template。第 48 行指定一个 anchor 标记，它具有 href 属性。href 属性的值引用 displayBook.py，并向其传递一个查询字符串。记住 allBooks.py 会将会话 ID 插入该查询字符串，从而实现会话跟踪机制。当前 product 的 isbn 使用 {isbn} 插入。最终的查询字符串使 displayBook 能识别客户并显示书籍。anchor 标记包含了文本，以及 XML 文档的 title 和 editionNumber 元素的值（第 49~50 行）。

XSLT 文档（图 23.10）指定一个链接样式表，即 styles.css（第 21~22 行）。发送给客户的所有 XHTML 文档都使用 styles.css 来保证格式的统一。图 23.11 列出了 styles.css 的内容。

```

1  body          { text-align: center;
2                  background-color: #b0c4de }
3  .bold         { font-weight: bold }
4  .bigFont      { font-family: helvetica, arial, sans-serif;
5                  font-weight: bold;
6                  font-size: 2em;
7                  color: #00008b }
8  .italic       { font-style: italic }
9  .right        { text-align: right }
10 table, th, td { border: 3px groove;
11                  padding: 5px }
12 table         { background-color: #6495ed;
13                  margin-left: auto;
14                  margin-right: auto }

```

图 23.11 styles.css 统一所有 XHTML 文档的格式

移植性提示 23.2 不同浏览器具有不同的 CSS 文档支持级别。

第 1~2 行使 body 元素中的文本居中,并将背景色设为钢青色,这个颜色表示成十六进制是#b0c4de。第 3 行定义 bold 类,为文本应用加粗字体样式。第 4~7 行定义 bigFont 类,它具有 4 项 CSS 属性。这些属性为访问该类的元素应用加粗的 Helvetica 字体。另外, bigFont 倍增字号,并将颜色设为深蓝(#00008b)。如果系统上未安装 Helvetica 字体,浏览器将使用 Arial 字体;如果 Arial 字体也没有,就使用默认字体 sans-serif。italic 类可为文本应用倾斜字体样式(第 8 行)。right 类则使文本右对齐(第 9 行)。第 10~11 行指出,所有 table、th(表头数据)和 td(表格数据)元素都应显示宽度为 3 个像素的凹线边框;而且在单元格的文本和单元格的边框之间,应使用 5 个像素的内部填充。第 12~14 行指出,所有 table 元素都应该有一个蓝色背景(表示成十六进制数字#6495ed),而且所有 table 元素都要在它们的左侧和右侧使用预定义页边距。页边距使表格在网页上居中。尽管样式表并没有为每个 XHTML 文档都应用所有样式,但使用一个样式表,程序员可快速和方便地修改应用程序的外观。

23.8 查看一本书的详细资料

在 allBooks.py 中选择一本书,会把用户重定向到 displayBook.py 脚本(图 23.12)。该程序从查询字符串提取 ISBN,并判断用户选择的是哪本书。然后,程序获得书的 XML 表示,并参照特定于浏览器的 XSLT 样式表(displayBook.xsl)来处理那个表示。结果发送给用户。

如未在查询字符串中指定 ISBN,程序将用户重定向到 error.py(第 17 行)。否则 displayBook.py 会载入会话(第 22 行)。如果成功,displayBook.py 从 session 变量获得 Book 的列表(第 27 行)。第 28 行将 session.data 的 bookToAdd 键设为 None,表明在 titles 变量所存储的 Book 列表中,没有发现指定的 ISBN。

```

1  #!c:\Python\python.exe
2  # Fig. 23.12: displayBook.py
3  # Retrieve one book's XML representation, convert
4  # to required format using browser-specific XSLT
5  # stylesheet and display results.
6
7  import cgi
8  import sys
9  import Session
10 from xml.xslt import Processor
11 from xml.dom.DOMImplementation import implementation
12
13 form = cgi.FieldStorage()
14
15 # ISBN has not been specified
16 if not form.has_key( "isbn" ):
17     Session.redirect( "error.py?message=No+ISBN+given" )
18     sys.exit()
19
20 # Load Session
21 try:
22     session = Session.Session()
23 except Session.SessionError, message: # invalid/no session ID
24     Session.redirect( "error.py?message=%s" % message )
25     sys.exit()
26
27 titles = session.data[ "titles" ] # get titles
28 session.data[ "bookToAdd" ] = None # book has not been found
29
30 # locate Book object for selected book
31 for book in titles:
32
33     if form[ "isbn" ].value == book.isbn:
34         session.data[ "bookToAdd" ] = book
35         break
36
37 # book has been found

```



```

38 if session.data[ "bookToAdd" ] is not None:
39
40     # get XML from selected book
41     document = implementation.createDocument( None, None, None )
42     document.appendChild( session.data[ "bookToAdd" ].getXML(
43         document ) )
44
45     # process XML against XSLT style sheet
46     processor = Processor.Processor()
47     style = open( session.data[ "agent" ] + "/displayBook.xml" )
48     processor.appendStylesheetString( style.read() % \
49         { session.data[ "ID" ], session.data[ "ID" ] } )
50     results = processor.runNode( document )
51     style.close()
52
53     # save Session data and display processed XML
54     pageData = session.data[ "content type" ] - results
55     session.saveSession()
56     print pageData
57 else:
58
59     # invalid ISBN has been specified
60     Session.redirect( "error.py?message=Nonexistent+ISBN" )

```

图 23.12 displayBook.py 将所选书籍的 XML 表示转换成特定于浏览器的格式

第 31~35 行遍历 titles，查找具有指定 ISBN 的 Book。如果存在具有指定 ISBN 的一本书，就将 session.data 的 bookToAdd 键设为相匹配的 Book 实例，并终止循环。如果没有任何一本书具有指定的 ISBN，循环仍会终止，只是 session.data 的 bookToAdd 键仍为 None 值。

第 38 行判断是否存在一本相匹配的书。如果存在匹配（即 session.data 的 bookToAdd 键不是 None），就执行第 40~56 行。第 41 行新建一个 XML 文档。第 42~43 行使用 appendChild 方法，将相匹配的 Book 的 product 元素追加到 Document。第 46~51 行根据一个特定于浏览器的 XSLT 样式表（displayBook.xml）对 XML 文档进行处理。样式表存储在当前目录的一个和客户类型对应的子目录中。注意程序必须先将会话 ID 插入样式表，然后才能处理 XML 文档。然后，我们保存会话，并向客户显示结果（第 54~56 行）。如果书籍不存在，第 60 行将客户重定向到 error.py。

图 23.13 展示 displayBook.xml 样式表。XML 文档的 6 个元素的值将放到最终生成的 XHTML 文档中。

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 23.13: displayBook.xml -->
4 <!-- XSLT style sheet that transforms XML generated by -->
5 <!-- displayBook.py into XHTML. -->
6
7 <xsl:stylesheet version = "1.0"
8     xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10     <xsl:output method = "xml" omit-xml-declaration = "no"
11         indent = "yes" doctype-system = "DTD/xhtml1-strict.dtd"
12         doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
13
14     <!-- specify the root of the XML document -->
15     <!-- that references this style sheet -->
16     <xsl:template match = "product">
17
18         <html xmlns = "http://www.w3.org/1999/xhtml">
19
20             <head>
21
22                 <!-- obtain book title from script to place in title -->
23                 <title><xsl:value-of select = "title" /></title>
24
25                 <link rel = "stylesheet" href = "/bookstore/styles.css"
26                     type = "text/css" />
27             </head>
28

```

```

29     <body>
30
31         <p class = "bigFont"><xsl:value-of select = "title" /></p>
32
33         <table>
34             <tr>
35
36                 <!-- create table cell for product image -->
37                 <td rowspan = "5"> <!-- cell spans 5 rows -->
38                     <img src = "/bookstore/images/{imageFile}"
39                         alt = "{title}" />
40                 </td>
41
42                 <!-- create table cells for price in row 1 -->
43                 <td class = "bold">Price:</td>
44
45                 <td><xsl:value-of select = "price" /></td>
46             </tr>
47
48             <tr>
49
50                 <!-- create table cells for ISBN in row 2 -->
51                 <td class = "bold">ISBN #:</td>
52
53                 <td><xsl:value-of select = "isbn" /></td>
54             </tr>
55
56             <tr>
57
58                 <!-- create table cells for edition in row 3 -->
59                 <td class = "bold">Edition:</td>
60
61                 <td><xsl:value-of select = "editionNumber" /></td>
62             </tr>
63
64             <tr>
65
66                 <!-- create table cells for copyright in row 4 -->
67                 <td class = "bold">Copyright:</td>
68
69                 <td><xsl:value-of select = "copyright" /></td>
70             </tr>
71
72             <tr>
73
74                 <!-- create Add to Cart button in row 5 -->
75                 <td>
76                     <form method = "post"
77                         action = "addToCart.py?ID=%s">
78                         <p><input type = "submit"
79                             value = "Add to Cart" /></p>
80                     </form>
81                 </td>
82
83                 <!-- create View Cart button in row 5 -->
84                 <td>
85                     <form method = "post"
86                         action = "viewCart.py?ID=%s">
87                         <p><input type = "submit"
88                             value = "View Cart" /></p>
89                     </form>
90                 </td>
91             </tr>
92         </table>
93
94     </body>
95
96 </html>
97
98 </xsl:template>
99

```

```
100 </xsl:stylesheet>
```

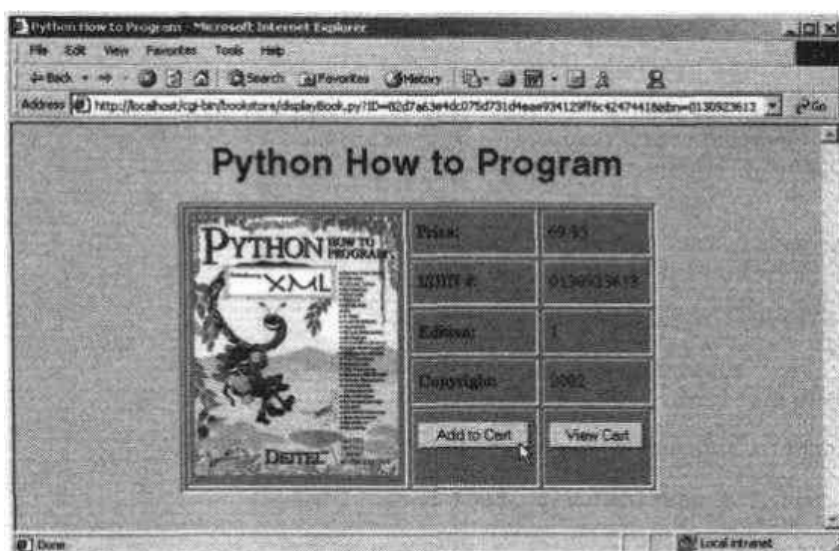


图 23.13 XSLT 样式表将一本书的 XML 表示转换成 XHTML 文档

第 23 行和第 31 行将书的 title 分别放到文档的 title 元素以及位于文档 body 元素起始处的一个段落中。第 38 行指定一个 img 元素，其中包含 XML 文档的 imageFile 元素的值。该元素指定了书的封面图像文件名。第 39 行使用书的 title 指定 img 元素的 alt 属性。第 45 行、第 53 行、第 61 行和第 69 行分别将书的 price、isbn、editionNumber 和 copyright 放到表格单元格中。第 76~80 行和第 85~89 行分别创建 Add to Cart 按钮 (addToCart.py) 和 View Cart (viewCart.py) 按钮。两个按钮都使用 post 表单方法将会话 ID 传给它们的目标文件。会话 ID 在第 77 行和第 86 行由 displayBook.py 插入。

23.9 在购物车中添加商品

在图 23.13 生成的文档中按下 Add to Cart (添加到购物车) 按钮，addToCart.py 程序就会更新购物车。CartItem 实例代表购物车中的商品。该类的一个实例负责将一件商品及其当前数量存储到购物车中。为了和我们的网上书店配合使用，一个 CartItem 要维护一个 Book 实例以及该 Book 在购物车中的数量。如果用户选择购物车中已经存在的一件商品，就在 CartItem 中更新那件商品的数量。否则，脚本将新建数量为 1 的一个 CartItem。更新了购物车后，用户被重定向到 viewCart.py 以查看购物车内容。图 23.14 和图 23.15 展示了 CartItem 类以及 addToCart.py 程序。

```
1 # Fig. 23.14: CartItem.py
2 # Maintains an item and a quantity.
3
4 class CartItem:
5     """Class that maintains an item and its quantity"""
6
7     def __init__( self, itemToAdd, number ):
8         """Initialize a CartItem"""
9
10        self.item = itemToAdd
11        self.quantity = number
```

图 23.14 CartItem 包含一个 item 及其 quantity

```
1 #!c:\Python\python.exe
2 # Fig. 23.15: addToCart.py
3 # Create new/update CartItem for selected Book object
4
5 import sys
```

```

6 import Session
7 import CartItem
8
9 # load Session
10 try:
11     session = Session.Session()
12 except Session.SessionError, message: # invalid/no session ID
13     Session.redirect( "error.py?message=%s" % message )
14     sys.exit()
15
16 book = session.data[ "bookToAdd" ]
17 cart = session.data[ "cart" ]
18
19 # if book is already in cart, update quantity
20 if book.isbn in cart.keys():
21     cartItem = cart[ book.isbn ]
22     cartItem.quantity += 1
23
24 # otherwise, create and add a new CartItem to cart
25 else:
26     cart[ book.isbn ] = CartItem.CartItem( book, 1 )
27
28 # update cart attribute
29 session.data[ "cart" ] = cart
30
31 # save Session data and send user to viewCart.py
32 nextPage = "viewCart.py?ID=%s" % session.data[ "ID" ]
33 session.saveSession()
34 Session.redirect( nextPage )

```

图 23.15 addToCart.py 将商品放到购物车，并调用 viewCart.py 来显示购物车内容

程序首先获得客户的 Session 实例（第 10~14 行）。如果不存在用于该客户的会话，客户会被重定向到 error.py（第 13 行）。否则，第 16 行获得 session.data 的 bookToAdd 键的值，即要添加到购物车的 Book。第 17 行获得 session.data 的 cart 键的值，即代表购物车的字典。第 20 行的 if 语句判断购物车是否已经包含指定的书籍。如果是，第 21 行就获得代表那本书的 CartItem，第 22 行则自增该 CartItem 的数量。否则，第 26 行新建一个 CartItem，将数量设为 1，再将商品放到购物车（书的 ISBN 作为键）。第 29 行将 session.data 的 cart 键设为更新过的字典（cart）。然后，第 32~34 行保存会话，并将用户转移到 viewCart.py，以显示购物车内容。

23.10 查看购物车

图 23.16 的 viewCart.py 程序从购物车提取 CartItems，计算购物车中每件商品的小计金额，计算车中全部商品的总计金额，并创建一个文档，使客户能以表格形式查看购物车。

首先载入会话（第 13~17 行）。如果出错，程序将用户重定向到 error.py（第 16 行）。第 19 行获得会话的购物车。第 20 行将变量 total 初始化为 0。接着新建一个 XML 文档，并为 Document 追加一个 cart 元素（第 23~25 行）。

```

1 #!c:\Python\python.exe
2 # Fig. 23.16: viewCart.py
3 # Generate XML representing cart, convert
4 # to required format using browser-specific XSLT
5 # style sheet and display results.
6
7 import sys
8 import Session
9 from xml.xslt import Processor
10 from xml.dom.DOMImplementation import implementation
11
12 # load Session
13 try:

```

```

14 session = Session.Session()
15 except Session.SessionError, message: # invalid/no session ID
16     Session.redirect( "error.py?message=%s" % message )
17     sys.exit()
18
19 cart = session.data[ "cart" ]
20 total = 0 # total for all ordered items
21
22 # generate XML representing cart object
23 document = implementation.createDocument( None, None, None )
24 cartNode = document.createElement( "cart" )
25 document.appendChild( cartNode )
26
27 # add XML representation for each cart item
28 for cartItem in cart.values():
29
30     # get book data, calculate subtotal and total
31     book = cartItem.item
32     quantity = cartItem.quantity
33     price = book.price
34     subtotal = quantity * price
35     total += subtotal
36
37     # create an orderProduct element
38     orderProduct = document.createElement( "orderProduct" )
39
40     # create a product element and append to orderProduct
41     productNode = book.getXML( document )
42     orderProduct.appendChild( productNode )
43
44     # create a quantity element and append to orderProduct
45     quantityNode = document.createElement( "quantity" )
46     quantityNode.appendChild( document.createTextNode( "%d" %
47         quantity ) )
48     orderProduct.appendChild( quantityNode )
49
50     # create a subtotal element and append to orderProduct
51     subtotalNode = document.createElement( "subtotal" )
52     subtotalNode.appendChild( document.createTextNode( "%.2f" %
53         subtotal ) )
54     orderProduct.appendChild( subtotalNode )
55
56     # append orderProduct to cartNode
57     cartNode.appendChild( orderProduct )
58
59 # set the total attribute of cart element
60 cartNode.setAttribute( "total", "%.2f" % total )
61
62 # make current total a session attribute
63 session.data[ "total" ] = total
64
65 # process generated XML against XSLT style sheet
66 processor = Processor.Processor()
67 style = open( session.data[ "agent" ] + "/viewCart.xsl" )
68 processor.appendStylesheetString( style.read() % \
69     ( session.data[ "ID" ], session.data[ "ID" ] ) )
70 results = processor.runNode( document )
71 style.close()
72
73 # save Session data and display processed XML
74 pageData = session.data[ "content type" ] + results
75 session.saveSession()
76 print pageData

```

图 23.16 viewCart.py 获得购物车并输出一个文档, 以表格形式显示购物车内容

第 28~57 行计算购物车中所有商品的总计金额。第 31~32 行从 CartItem 中获取 Book 实例和 quantity (数量)。第 33 行获得 Book 的 price。第 34 行计算 CartItem 的小计金额。第 35 行更新所有购物车商品的总计金额。第 38 行为购物车中的每件商品都创建一个 XML 的 orderProduct 元素。

每个 orderProduct 元素都包含 3 个子元素: product、quantity 和 subtotal。首先获取当前 Book 的 product 元素 (第 41 行)。第 42 行调用 appendChild 方法, 把这个 product 元素添加到 orderProduct 元素。接着, 第 45~48 行创建并追加 quantity 元素。注意程序首先必须将当前 CartItem 的数量转换成一个字符串, 然后才能创建 quantity 元素。第 51~54 行创建并追加 orderProduct 的 subtotal 元素。subtotal 元素包含了当前 CartItem 的小计金额, 它格式化成为两个小数位。第 57 行将当前 orderProduct 追加到 cart 元素。

为每种 CartItem 都创建一个 orderProduct 元素, 并把它追加到 cart 元素后, 程序接着设置 cart 元素的 total 属性 (第 60 行)。第 63 行将当前销售总金额 (即所有小计金额的和) 存储到会话字典, 并用 total 指定具体的键。第 66~71 行参照一个特定于浏览器的 XSLT 样式表 (viewCart.xml) 来处理 XML 文档。处理之前, 会话 ID 必须插入样式表。第 74~76 行保存会话, 并向客户显示转换好的 XML。

图 23.17 显示了 XSLT 转换过程中使用的 viewCart.xml 样式表。

```

1  <?xml version = "1.0"?>
2
3  <!-- Fig. 23.17: viewCart.xml -->
4  <!-- XSLT style sheet that transforms XML generated by -->
5  <!-- viewCart.py into XHTML. -->
6
7  <xsl:stylesheet version = "1.0"
8      xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10     <xsl:output method = "xml" omit-xml-declaration = "no"
11         indent = "yes" doctype-system = "DTD/xhtml1-strict.dtd"
12         doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
13
14     <xsl:template match = "cart">
15
16         <html xmlns = "http://www.w3.org/1999/xhtml">
17
18             <head>
19                 <title>Your Online Shopping Cart</title>
20                 <link rel = "stylesheet" href = "/bookstore/styles.css"
21                     type = "text/css" />
22             </head>
23
24             <body>
25
26                 <p class = "bigFont">Shopping Cart</p>
27
28                 <xsl:choose>
29                     <xsl:when test = "@total = '0.00'">
30                         <p class = "bold">
31                             Your shopping cart is currently empty.</p>
32                     </xsl:when>
33
34                     <xsl:otherwise> <!-- total != 0.00 -->
35                         <table class = "cart">
36                             <tr>
37                                 <th>Product</th>
38                                 <th>Quantity</th>
39                                 <th>Price</th>
40                                 <th>Total</th>
41                             </tr>
42
43                             <xsl:apply-templates select = "orderProduct">
44
45                                 <!-- sort orderProducts by product/title -->
46                                 <xsl:sort select = "product/title" />
47
48                             </xsl:apply-templates>
49
50                             <tr>
51                                 <td colspan = "4" class = "bold right">
52                                     Total: <xsl:value-of select = "@total" />
53                                 </td>
54                             </tr>

```

```

55         </table>
56
57         </xsl:otherwise>
58     </xsl:choose>
59
60     <p class = "bold green">
61         <a href = "allBooks.py?ID=%s">Continue Shopping</a>
62     </p>
63
64     <form method = "post" action = "order.py?ID=%s">
65         <p><input type = "submit" value = "Check Out" /></p>
66     </form>
67
68 </body>
69
70 </html>
71
72 </xsl:template>
73
74 <xsl:template match = "orderProduct">
75
76     <tr>
77         <td><xsl:value-of select = "product/title" />,
78         <xsl:value-of select = "product/editionNumber" /></td>
79         <td><xsl:value-of select = "quantity" /></td>
80         <td class = "right"><xsl:value-of select =
81             "product/price" /></td>
82         <td class = "bold right"><xsl:value-of select =
83             "subtotal" /></td>
84     </tr>
85
86 </xsl:template>
87
88 </xsl:stylesheet>

```



图 23.17 将 XML 形式的购物车转换为 XHTML 文档的 XSLT 样式表

第一个 `xsl:template` (第 14~72 行) 匹配 `cart` 元素。第 28 行开始一个 `xsl:choose` 元素。如果 `cart` 的 `total` 属性 (用 `@` 字符指示) 等于 "0.00", 程序向客户显示一条消息, 指出购物车目前是空的 (第 30~31 行); 否则, 就为购物车中的所有商品创建一个表格 (第 34~57 行)。

第 43~48 行将所有匹配项插入 `orderProduct` 模板, 并按照它们的 `product/title` 元素排序。`orderProduct` 模板 (第 74~86 行) 对 `orderProduct` 元素进行匹配 (即描述如何转换)。第 77~83 行在表格单元格中插入 `orderProduct` 的 `product/title`、`product/editionNumber`、`quantity`、`product/price` 和 `subtotal`。然后, 第 50~54 行插入一个表格行, 显示全部商品总计金额。接着为用户创建两个选项。第一个是指向 `allBooks.py` 的超链接 (第 61 行), 第二个是 `Check Out` (结账) 按钮, 将用户重定向到 `order.py` (第 64~66 行)。

23.11 结账

查看购物车时，用户可单击 Check Out（结账）按钮以执行 `order.py`，如图 23.18 所示。该脚本获取名为 `orderForm` 的一个静态页，它的内容对于不同客户类型也是不同的。文件存储在与客户类型对应的一个子目录中（例如对于 XHTML 客户，是 `xhtml/orderForm.html`）。`orderForm` 表单请求用户输入他们的姓名、地址、电话号码和信用卡信息。表单帮助结束应用程序。对于本例，该表单无任何实际用途。通常，表单元素要进行一些服务器端校验（例如验证信用卡是否有效）。用户提交表单后，会被重定向到 `process.py`，以完成书籍订单。

第 9~13 行首先载入会话。如果出错，程序将用户重定向到 `error.py`（第 12 行）。第 16~17 行打开特定于浏览器订单表单。注意目录名不一定和文件扩展名对应（例如，XHTML Basic 客户使用 `xhtml_basic` 目录，文件扩展名是 `html`）。第 18~19 行创建字符串，其中包含客户的内容类型头（即包含在 `contentType.txt` 中的头），以及 `orderForm` 的内容，并用会话 ID 进行格式化。第 23~24 行保存会话，并显示订单。图 23.19 显示 `orderForm.html`，它是由 `order.py` 显示给 XHTML 客户的订单表单。

```

1  #!c:\Python\python.exe
2  # Fig. 23.18: order.py
3  # Display order form to get information from customer
4
5  import sys
6  import Session
7
8  # load Session
9  try:
10     session = Session.Session()
11 except Session.SessionError, message: # invalid/no session ID
12     Session.redirect( "error.py?message=%s" % message )
13     sys.exit()
14
15 # display content type and orderForm for specific client-type
16 content = open( "%s/orderForm.%s" % ( session.data[ "agent" ],
17     session.data[ "extension" ] ) )
18 pageData = session.data[ "content type" ] + content.read() % \
19     session.data[ "ID" ]
20 content.close()
21
22 # save Session data and display order form
23 session.saveSession()
24 print pageData

```

图 23.18 `order.py` 为客户获取、格式化和显示静态订单表单页

```

1  <!-- Fig. 23.19: orderForm.html -->
2  <!-- Static XHTML to be displayed by order.py -->
3
4  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
5     "DTD/xhtml11-strict.dtd">
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8
9     <head>
10         <title>Order</title>
11         <link rel = "stylesheet" href = "/bookstore/styles.css"
12             type = "text/css" />
13     </head>
14
15     <body>
16
17         <p class = "bigFont">Shopping Cart Check Out</p>
18
19         <!-- form to input user information and credit card. -->
20         <!-- note: no need to input real data in this example. -->
21         <form method = "post" action = "process.py?ID=%s">

```



```

22     <p style = "font-weight: bold">
23         Please input the following information</p>
24
25     <table>
26         <tr>
27             <td class = "right bold">First name:</td>
28
29             <td>
30                 <input type = "text" name = "firstname"
31                     size = "25" />
32             </td>
33         </tr>
34         <tr>
35             <td class = "right bold">Last name:</td>
36
37             <td>
38                 <input type = "text" name = "lastname"
39                     size = "25" />
40             </td>
41         </tr>
42         <tr>
43             <td class = "right bold">Street:</td>
44
45             <td>
46                 <input type = "text" name = "street"
47                     size = "25" />
48             </td>
49         </tr>
50         <tr>
51             <td class = "right bold">City:</td>
52
53             <td>
54                 <input type = "text" name = "city"
55                     size = "25" />
56             </td>
57         </tr>
58         <tr>
59             <td class = "right bold">State:</td>
60
61             <td>
62                 <input type = "text" name = "state"
63                     size = "2" />
64             </td>
65         </tr>
66         <tr>
67             <td class = "right bold">Zip code:</td>
68
69             <td>
70                 <input type = "text" name = "zipcode"
71                     size = "10" />
72             </td>
73         </tr>
74         <tr>
75             <td class = "right bold">Phone #:</td>
76
77             <td>
78                 <input type = "text" name = "phone"
79                     size = "12" />
80             </td>
81         </tr>
82         <tr>
83             <td class = "right bold">Credit Card #:</td>
84
85             <td>
86                 <input type = "text" name = "creditcard"
87                     size = "25" />
88             </td>
89         </tr>
90         <tr>
91             <td class = "right bold">Expiration (mm/yyyy):</td>
92

```

```

93         <td>
94             <input type = "text" name = "expires"
95                 size = "2" />
96
97             <input type = "text" name = "expires2"
98                 size = "4" />
99         </td>
100     </tr>
101 </table>
102
103     <p><input type = "submit" value = "Submit" /></p>
104
105 </form>
106
107 </body>
108
109 </html>

```



图 23.19 orderForm.html 是由 order.py 为 XHTML 客户显示的订单表单

23.12 处理订单

图 23.20 (process.py) 模拟处理用户的信用卡信息，并获取特定于浏览器的一个名为 thankYou 的文档，这是一个静态网页（第 16~17 行）。具体文件存储在和客户类型对应的一个子目录中（例如对于 XHTML 客户，是 xhtml/thankYou.html）。然后，程序将最终总计金额插入 thankYou 的内容中（第 18~19 行），并向客户显示此页。我们的书店应用程序并不执行真正的信用卡处理，所以交易现已完成。第 23 行调用 Session 的 deleteSession 方法，删除用于当前客户的会话文件。在真正的商店中，除非信用卡公司确认了交易，否则会话是无效的。图 23.21 展示了用于 XHTML 客户的 thankYou 文件。

```

1 #!c:\Python\python.exe
2 # Fig. 23.20: process.py
3 # Display thank you page to customer and delete session
4
5 import sys
6 import Session
7
8 # load session
9 try:
10     session = Session.Session()

```

```

11 except Session.SessionError, message: # invalid/no session ID
12     Session.redirect( "error.py?message=%s" % message )
13     sys.exit()
14
15 # display content type and thankYou for specific client-type
16 content = open( "%s/thankYou.%s" % ( session.data[ "agent" ],
17     session.data[ "extension" ] ) )
18 pageData = session.data[ "content type" ] + content.read() % \
19     session.data[ "total" ]
20 content.close()
21
22 # delete session and display thank you page
23 session.deleteSession()
24 print pageData

```

图 23.20 process.py 获取、格式化和显示静态感谢页

```

1 <!-- Fig. 23.21: thankYou.html -->
2 <!-- Static XHTML to be displayed by process.py -->
3
4 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
5     "DTD/xhtml11-strict.dtd">
6
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8
9     <head>
10         <title>Thank You!</title>
11         <link rel = "stylesheet" href = "/bookstore/styles.css"
12             type = "text/css" />
13     </head>
14
15     <body>
16
17         <p class = "bigFont">Thank You</p>
18         <p>Your order has been processed.</p>
19         <p>Your credit card has been billed:
20             <span class = "bold">$.2f</span>
21         </p>
22
23     </body>
24
25 </html>

```



图 23.21 thankYou.html 是 process.py 为 XHTML 客户显示的退出页面

23.13 错误处理

如果网上书店应用程序出错, 用户会被重定向到 error.py (图 23.22), 它向客户显示一条错误消息。

```

1 #!c:\Python\python.exe
2 # Fig. 23.22: error.py
3 # Generate XML error message and display to user
4 # using a browser-specific XSLT style sheet.
5

```

```

6 import cgi
7 import Session
8 from xml.xslt import Processor
9 from xml.dom.DOMImplementation import implementation
10
11 form = cgi.FieldStorage()
12
13 if form.has_key( "message" ):
14
15     # create DOM for error message
16     document = implementation.createDocument( None, None, None )
17     error = document.createElement( "error" )
18     message = document.createElement( "message" )
19     message.appendChild( document.createTextNode(
20         form[ "message" ].value ) )
21     error.appendChild( message )
22     document.appendChild( error )
23
24     # process against XSLT style sheet
25     processor = Processor.Processor()
26     style = open( Session.getClientType()[ 0 ] + "/error.xsl" )
27     processor.appendStylesheetStream( style )
28     results = processor.runNode( document )
29     style.close()
30
31     # display content type and processed XML
32     print Session.getContentType() + results

```

图 23.22 error.py 显示动态创建的错误页面

如果在查询字符串中指定了一条错误消息，就首先创建一个新的 XML 文档（第 16 行）。第 17~18 行创建 `error` 和 `message` 元素。第 19~20 行将指定的错误消息追加到 `message` 元素。第 21 行将 `message` 元素追加到 `error` 元素。第 22 行将 `error` 元素追加到 XML 文档。

第 25~29 行根据一个特定于浏览器的 XSLT 样式表 (`error.xsl`) 来处理文档。由于 `error.py` 没有会话，所以必须调用 `Session` 的 `getClientType` 和 `getContentType` 函数，以确定具体要使用的文件。结果向用户显示出来（第 32 行）。

图 23.23 展示了在针对 XHTML 客户的 XSLT 转换中使用的 `error.xsl` 样式表文件。第 14~35 行定义一个 `xsl:template`，它对 `error` 元素进行匹配。第 28 行将 `message` 元素值插入一个 `paragraph` 标记。

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 23.23: error.xsl -->
4 <!-- XSLT style sheet that transforms XML generated by -->
5 <!-- error.py into XHTML. -->
6
7 <xsl:stylesheet version = "1.0"
8     xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10     <xsl:output method = "xml" omit-xml-declaration = "no"
11         indent = "yes" doctype-system = "DTD/xhtml1-strict.dtd"
12         doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
13
14     <xsl:template match = "error">
15
16         <html xmlns = "http://www.w3.org/1999/xhtml">
17
18             <head>
19                 <title>Error</title>
20                 <link rel = "stylesheet" href = "/bookstore/styles.css"
21                     type = "text/css" />
22             </head>
23
24             <body>
25
26                 <p class = "bigFont">Error message:</p>
27                 <p class = "bold">

```

```

28         <xsl:value-of select = "message" />
29     </p>
30
31 </body>
32
33 </html>
34
35 </xsl:template>
36
37 </xsl:stylesheet>

```

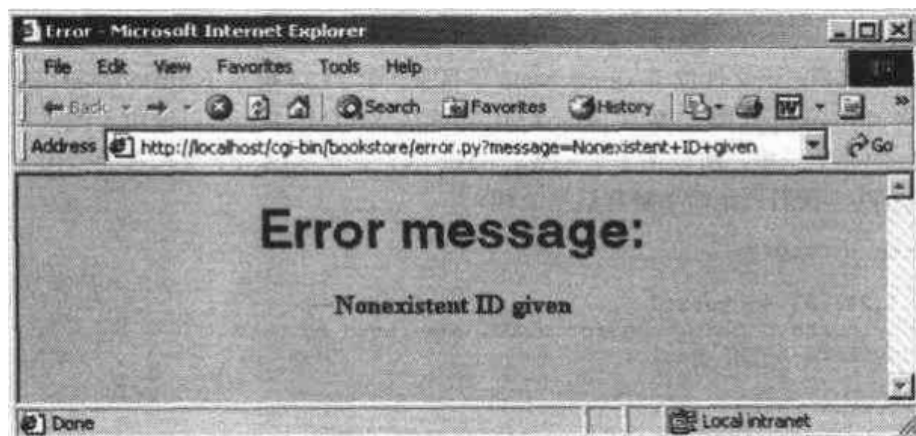


图 23.23 将错误的 XML 表示转换成 XHTML 文档的 XSLT 样式表

23.14 处理无线客户端（XHTML Basic 和 WML）

本节扩展书店应用程序以执行手持无线客户。“无线客户”包括 PDA（个人数字助理）和数字手机，它们需要支持来自“有线客户”（例如 Internet Explorer）的不同标记语言。本节演示了两种运用得最广泛的无线标记语言——XHTML Basic 和 WML。

WAP（无线应用协议）是无线设备和 Internet 之间的一种全球性通信标准，同时支持 XHTML Basic 和 WML。WAP 2.0 是目前最新版本，它规定用 XHTML Basic（XHTML 的一个子集）取代 WML，作为无线设备 Web 内容的标记语言。但是，由于许多设备还不支持 XHTML Basic，所以在我们的网上书店案例分析中，将同时支持两种标记语言。要了解 WAP 的详情，请访问 WAP Forum 网站（www.wapforum.org）。

在后续的小节，我们使用名为“微型浏览器”的一种软件来显示输出。微型浏览器适合在带宽和内存有限的情况下使用。针对不同类型的设备，有不同的微型浏览器可供选择。例如，Openwave 的 Mobile Browser 类似于 Microsoft Internet Explorer，只是要在无线设备上运行。

本节演示如何对网上书店系统进行扩展。事实上，只需新建支持特定标记语言的 XSLT 样式表，即可实现对新的客户类型的支持。

23.14.1 XHTML Basic 概述

XHTML Basic 是万维网协会（W3C）制定的一项标准，宗旨是为无线设备和内存容量有限的其他小型设备提供通用的标记语言。类似于 WML，XHTML Basic 也是从 XML（可扩展标记语言）派生出来的。作为 XHTML 的一个精简版本，XHTML Basic 可以很好地适应无线设备有限的内存容量。XHTML Basic 取消了不适合无线客户的一些特性，比如嵌套表格。此外，XHTML Basic 利用了 XML 严格的结构规则及其扩展能力。

XHTML Basic 用一个文本编辑器（比如记事本、vi 或 emacs）创建，并采用.html 或.htm 文件扩展名。

本节使用 Pixa Internet Microbrowser 来呈现 XHTML Basic 文档。Pixa 公司致力于为手机制造商开发软件平台和服务。Pixa Internet Microbrowser 模拟一个支持 Internet 连接的手机界面。在以后的例子中，Pixa Internet Microbrowser 将呈现 XHTML Basic 文档。Pixa Internet Microbrowser 2.1 可从 www.pixa.com 免费下载。具体安装指南请访问 Deitel & Associates 网站 (www.deitel.com)。

图 23.9 中，allBooks.py 的第 78 行打开了一个版本的 allBooks.xml，它存储在与客户类型对应的目录中：

```
style = open( session.data[ "agent" ] + "/allBooks.xml" )
```

对于 XHTML Basic 客户，该文件位于 xhtml_basic 子目录。图 23.24 展示了 allBooks.xml 的 XHTML Basic 版本。该文档可将 allBooks.py(图 23.9)生成的 XML 转换成 XHTML Basic，以便由无线客户呈现。XHTML Basic 是 XHTML 的一个子集，所以用于 XHTML Basic 客户的所有 XSLT 文档都类似用于 XHTML 客户的 XSLT 文档。所以，我们只是简单解释这些文档。

```

1  <?xml version = "1.0"?>
2
3  <!-- Fig. 23.24: allBooks.xml -->
4  <!-- XSLT style sheet that transforms XML generated by -->
5  <!-- books.py into XHTML Basic. -->
6
7  <xsl:stylesheet version = "1.0"
8    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10   <xsl:output method = "xml" omit-xml-declaration = "no"
11     indent = "yes" doctype-system =
12       "http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd"
13     doctype-public = "-//W3C//DTD XHTML Basic 1.0//EN"/>
14
15   <!-- template for catalog element -->
16   <xsl:template match = "catalog">
17
18     <html xmlns = "http://www.w3.org/1999/xhtml">
19
20       <head>
21         <title>Book List</title>
22         <link rel = "stylesheet" href = "/bookstore/styles.css"
23           type = "text/css" />
24       </head>
25
26       <body>
27
28         <p class = "bigFont">Available Books</p>
29         <p class = "bold">
30           Click a link to view book information</p>
31
32         <!-- match product elements to product template -->
33         <xsl:apply-templates select = "/catalog/product">
34
35           <!-- sort products by title -->
36           <xsl:sort select = "title" />
37
38         </xsl:apply-templates>
39
40       </body>
41     </html>
42   </xsl:template>
43
44   <!-- template for building row of Product information -->
45   <xsl:template match = "product">
46
47     <p><a href = "displayBook.py?ID=%s&isbn={isbn}">
48       <strong><xsl:value-of select = "title" />, <xsl:value-of
49         select = "editionNumber" /></strong>

```

```

52     </a></p>
53
54 </xsl:template>
55
56 </xsl:stylesheet>

```



图 23.24 用于 XHTML Basic 客户的 allBooks.xml

第 10~13 行输出了与 XHTML Basic 语法保持一致所需的行。xsl:template 定义 catalog 元素（第 16~44 行）。在这个模板中，我们插入 product 模板的任何匹配项（第 33~38 行）。这些匹配项根据它们的 title 元素进行排序（第 36 行），保证书籍按标题的字母顺序排列。

第 47~54 行为 product 元素定义一个 xsl:template。第 49 行指定一个具有 href 属性的 anchor 标记。href 属性值引用 displayBook.py，并在查询字符串中包含会话 ID 和 XML 文档的 isbn 元素的值。注意，allBooks.py 的第 71 行将会话 ID 插入这个查询字符串。当前 product 的 isbn 是用 {isbn} 插入的。最终的查询字符串确保 displayBook 能识别客户并正确地显示书籍。anchor 标记包含文本，以及 XML 文档的 title 和 editionNumber 元素的值（第 50~51 行）。

图 23.25 展示了用于 XHTML Basic 客户的 displayBook.xml。该文档将 displayBook.py（图 23.12）生成的 XML 转换成 XHTML Basic，使无线客户能够正确呈现它们。

第 24 行将书的 title 放到文档的 title 元素，第 32 行将 title 放到文档 body 元素起始处的一个段落。第 34 行指定一个 img 元素，其中包含 XML 文档的 imageFile 元素的值。该元素指定了书的封面图像文件名。注意对于 XHTML Basic 客户，我们使用的是存储在子目录 small_images 中的图像。内容虽然和较大的版本相同，但它们的大小进行了调整，以便在无线设备上显示。第 35 行使用书的 title 指定 img 元素的 alt 属性。第 39 行、第 42 行、第 45 行和第 48 行分别显示书的 price、isbn、editionNumber 和 copyright。第 52~55 行和第 60~63 行分别创建 Add to Cart 按钮（addToCart.py）和 View Cart 按钮（viewCart.py）。两个按钮都用 post 表单方法将会话 ID 传给它们的目标文件。

图 23.26 展示了用于 XHTML Basic 客户的 viewCart.xml。该文档将 viewCart.py（图 23.16）生成的

XML 转换成可由客户正确呈现的 XHTML Basic。

```

1  <?xml version = "1.0"?>
2
3  <!-- Fig. 23.25: displayBook.xml -->
4  <!-- XSLT style sheet that transforms XML generated by -->
5  <!-- displayBook.py into XHTML Basic. -->
6
7  <xsl:stylesheet version = "1.0"
8      xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10     <xsl:output method = "xml" omit-xml-declaration = "no"
11         indent = "yes" doctype-system =
12             "http://www.w3.org/TR/xhtml1-basic/xhtml1-basic10.dtd"
13         doctype-public = "-//W3C//DTD XHTML Basic 1.0//EN"/>
14
15     <!-- specify the root of the XML document -->
16     <!-- that references this style sheet -->
17     <xsl:template match = "product">
18
19         <html xmlns = "http://www.w3.org/1999/xhtml">
20
21             <head>
22
23                 <!-- obtain book title from script to place in title -->
24                 <title><xsl:value-of select = "title" /></title>
25
26                 <link rel = "stylesheet" href = "/bookstore/styles.css"
27                     type = "text/css" />
28             </head>
29
30             <body>
31                 <p class = "bigFont">
32                     <xsl:value-of select = "title" /></p>
33
34                 <img src = "/bookstore/small_images/{imageFile}"
35                     alt = "{title}" />
36
37                 <!-- show price, ISBN, edition and copyright -->
38                 <p class = "bold">Price:
39                     <xsl:value-of select = "price" /></p>
40
41                 <p class = "bold">ISBN:
42                     <xsl:value-of select = "isbn" /></p>
43
44                 <p class = "bold">Edition:
45                     <xsl:value-of select = "editionNumber" /></p>
46
47                 <p class = "bold">Copyright:
48                     <xsl:value-of select = "copyright" /></p>
49
50                 <!-- create Add to Cart button -->
51                 <p>
52                     <form method = "post" action = "addToCart.py?ID=%s">
53                         <p><input type = "submit"
54                             value = "Add to Cart" /></p>
55                     </form>
56                 </p>
57
58                 <!-- create View Cart button -->
59                 <p>
60                     <form method = "post" action = "viewCart.py?ID=%s">
61                         <p><input type = "submit"
62                             value = "View Cart" /></p>
63                     </form>
64                 </p>
65
66             </body>
67
68         </html>

```



```

69
70 </xsl:template>
71
72 </xsl:stylesheet>

```



图 23.25 用于 XHTML Basic 客户的 displayBook.xsl

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 23.26: viewCart.xsl -->
4 <!-- XSLT style sheet that transforms XML generated by -->
5 <!-- viewCart.py into XHTML Basic. -->
6
7 <xsl:stylesheet version = "1.0"
8   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10  <xsl:output method = "xml" omit-xml-declaration = "no"
11    indent = "yes" doctype-system =
12      "http://www.w3.org/TR/xhtml1-basic/xhtml1-basic10.dtd"
13    doctype-public = "-//W3C//DTD XHTML Basic 1.0//EN"/>
14
15  <xsl:template match = "cart">
16
17    <html xmlns = "http://www.w3.org/1999/xhtml">
18
19      <head>
20        <title>Shopping Cart</title>
21        <meta name = "PixoFriendly" content = "true" />
22        <link rel = "stylesheet" href = "/bookstore/styles.css"
23          type = "text/css" />
24      </head>
25
26      <body>
27
28        <p class = "bigFont">Shopping Cart</p>
29
30        <xsl:choose>
31          <xsl:when test = "@total = '0.00'">
32            <p class = "bold">
33              Your shopping cart is currently empty.</p>
34          </xsl:when>
35
36          <xsl:otherwise> <!-- total != 0.00 -->

```

```

37
38         <table border = "true">
39             <xsl:apply-templates select = "orderProduct">
40
41                 <!-- sort orderProducts by product/title -->
42                 <xsl:sort select = "product/title" />
43
44             </xsl:apply-templates>
45         </table>
46
47         <p class = "bold right">Total: <xsl:value-of
48             select = "@total" /></p>
49
50     </xsl:otherwise>
51 </xsl:choose>
52
53     <p class = "bold green">
54         <a href = "allBooks.py?ID=%s">Continue Shopping</a>
55     </p>
56
57     <form method = "post" action = "order.py?ID=%s">
58         <p><input type = "submit" value = "Check Out" /></p>
59     </form>
60
61 </body>
62
63 </html>
64
65 </xsl:template>
66
67 <xsl:template match = "orderProduct">
68
69     <tr><td><ul>Product:<br />
70         <xsl:value-of select = "product/title" />,
71         <xsl:value-of select = "product/editionNumber" /><br />
72         Quantity: <xsl:value-of select = "quantity" /><br />
73         Price: <xsl:value-of select = "product/price" /><br />
74         Subtotal: <xsl:value-of select = "subtotal" /><br />
75     </ul></td></tr>
76
77 </xsl:template>
78
79 </xsl:stylesheet>

```



图 23.26 用于 XHTML Basic 客户的 viewCart.xsl

第一个 `xsl:template` (第 15~65 行) 匹配 `cart` 元素。第 21 行指定一个 `meta` 标记, 名称是 "PixoFriendly", 内容是 "true"。尽管 XHTML Basic 客户通常不接受表格, 但这一行使表格能在 Pixo Internet Microbrowser 中正确显示。第 30 行开始一个 `xsl:choose` 元素。如果 `cart` 的 `total` 属性 (用 @ 注明) 等于 "0.00", 程序向客户显示一条消息, 指出购物车目前是空的 (第 32~33 行); 否则, 就为购物车中的所有商品创建一个表格 (第 36~50 行)。

第 39~44 行将所有匹配项插入 `orderProduct` 模板, 并按它们的 `product/title` 元素排序。`orderProduct` 模板 (第 67~77 行) 对 `orderProduct` 元素进行匹配。第 70~74 行在一个表格单元格中插入 `orderProduct` 的 `product/title`、`product/editionNumber`、`quantity`、`product/price` 和 `subtotal` (为了在 XHTML Basic 客户上清楚地显示, 我们只为每个 `orderProduct` 使用一个单元格)。然后, 第 47~48 行显示全部商品总计金额。第 53~59 行为用户创建两个选项。第一个是指向 `allBooks.py` 的超链接 (第 54 行), 第二个是 Check Out (结账) 按钮, 它可将用户重定向到 `order.py` (第 57~59 行)。

图 23.27 展示了用于 XHTML Basic 客户的 `orderForm.html`, 它是由 `order.py` (图 23.18) 向 XHTML Basic 客户显示的订单表单。图 23.28 展示了 `thankYou.html`, 由 `process.py` (图 23.20) 向客户显示。图 23.29 展示了 `error.xsl`, 它将 `error.py` (图 23.22) 生成的 XML 转换成 XHTML Basic 格式。第 15~37 行定义一个 `xsl:template`, 以便匹配 `error` 元素。第 30 行将 `message` 元素值插入一个 `paragraph` 标记。

```

1  <!-- Fig. 23.27: orderForm.html -->
2  <!-- Static XHTML Basic to be displayed by order.py -->
3
4  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
5    "http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd">
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8
9    <head>
10      <title>Order</title>
11      <meta name = "PixoFriendly" content = "true" />
12      <link rel = "stylesheet" href = "/bookstore/styles.css"
13        type = "text/css" />
14    </head>
15
16    <body>
17
18      <p class = "bigFont">Shopping Cart Check Out</p>
19
20      <!-- form to input user information and credit card -->
21      <!-- note: no need to input real data in this example -->
22      <form method = "post" action = "process.py?ID=%s">
23        <p style = "font-weight: bold">
24          Input the following:</p>
25
26        <table border = "true">
27          <tr>
28            <td class = "right bold">First name:<br />
29              <input type = "text" name = "firstname"
30                size = "25" />
31            </td>
32          </tr>
33          <tr>
34            <td class = "right bold">Last name:<br />
35              <input type = "text" name = "lastname"
36                size = "25" />
37            </td>
38          </tr>
39          <tr>
40            <td class = "right bold">Street:<br />
41              <input type = "text" name = "street"
42                size = "25" />
43            </td>
44          </tr>
45          <tr>
46            <td class = "right bold">City:<br />

```

```

47         <input type = "text" name = "city"
48             size = "25" />
49     </td>
50 </tr>
51 <tr>
52     <td class = "right bold">State:
53         <input type = "text" name = "state"
54             size = "2" />
55     </td>
56 </tr>
57 <tr>
58     <td class = "right bold">Zip code:
59         <input type = "text" name = "zipcode"
60             size = "10" />
61     </td>
62 </tr>
63 <tr>
64     <td class = "right bold">Phone #:
65         <input type = "text" name = "phone"
66             size = "12" />
67     </td>
68 </tr>
69 <tr>
70     <td class = "right bold">Credit Card #:<br />
71         <input type = "text" name = "creditcard"
72             size = "25" />
73     </td>
74 </tr>
75 <tr>
76     <td class = "right bold">Expiration (mm/yyyy):
77         <br />
78         <input type = "text" name = "expires"
79             size = "2" />
80         <input type = "text" name = "expires2"
81             size = "4" />
82     </td>
83 </tr>
84 </table>
85
86 <p><input type = "submit" value = "Submit" /></p>
87
88 </form>
89
90 </body>
91
92 </html>

```



图 23.27 用于 XHTML Basic 客户的 orderForm.html

```

1 <!-- Fig. 23.28: thankYou.html -->
2 <!-- Static XHTML Basic to be displayed by process.py -->
3
4 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
5   "http://www.w3.org/TR/xhtml1-basic/xhtml1-basic10.dtd">
6
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8
9   <head>
10     <title>Thank You!</title>
11     <link rel = "stylesheet" href = "/bookstore/styles.css"
12         type = "text/css" />
13   </head>
14
15   <body>
16
17     <p class = "bigFont">Thank You</p>
18     <p>Your order has been processed.</p>
19     <p>Your credit card has been billed:
20       <span class = "bold">$$.2f</span>
21     </p>
22
23   </body>
24
25 </html>

```



图 23.28 用于 XHTML Basic 客户的 thankYou.html

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 23.29: error.xsl -->
4 <!-- XSLT style sheet that transforms XML generated by -->
5 <!-- error.py into XHTML Basic. -->
6
7 <xsl:stylesheet version = "1.0"
8   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10   <xsl:output method = "xml" omit-xml-declaration = "no"
11     indent = "yes" doctype-system =
12       "http://www.w3.org/TR/xhtml1-basic/xhtml1-basic10.dtd"
13     doctype-public = "-//W3C//DTD XHTML Basic 1.0//EN"/>
14
15   <xsl:template match = "error">
16
17     <html xmlns = "http://www.w3.org/1999/xhtml">
18

```

```

19     <head>
20         <title>Error</title>
21         <link rel = "stylesheet" href = "/bookstore/styles.css"
22             type = "text/css" />
23     </head>
24
25     <body>
26
27         <p class = "bigFont">Error message:</p>
28
29         <p class = "bold">
30             <xsl:value-of select = "message" />
31         </p>
32
33     </body>
34
35 </html>
36
37 </xsl:template>
38
39 </xsl:stylesheet>

```



图 23.29 用于 XHTML Basic 客户的 error.xsl

23.14.2 WML 概述

本节介绍用 WML（无线标记语言）进行的无线编程。WML 类似于 XHTML Basic，也是用于标识文档元素的一种标记语言，使无线设备能够呈现文档。WML 之所以是无线设备的理想选择，是因为它能适应它们有限的内存容量和狭窄的显示屏。我们采用与 WAP 标准相符的方式使用 WML。

每个 WML 文档都称为一个“卡片组”（Deck）；每个卡片组都包含一个或多个页，它们称为“卡片”（Card）。卡片是 WML 文档的可呈现单元，适合用于屏幕大小有限的 WAP 客户（即支持 WAP 的任何设备）。每个卡片都可包含文本内容及导航控件，以简化用户交互。虽然每次只能查看一个卡片，但可在不同卡片之间快速切换，因为整个卡片组都存储在微型浏览器中。

WML 文档用文本编辑器创建（记事本、vi 和 emacs 等），采用.wml 扩展名。本节使用 Openwave Simulator（www.openwave.com）呈现 WML 文档。Openwave Simulator 是 Openwave SDK WAP Edition 的

一部分, 可从 developer.openwave.com/download/license_50.html 免费下载。要了解具体安装指南, 请访问 www.deitel.com。

图 23.30 展示了 allBooks.xsl 用于 WML 客户的版本。该文档位于 wml 子目录中, 可将 allBooks.py (图 23.9) 生成的 XML 内容转换成 WML 格式, 以便由无线客户正确呈现。

第 10~13 行输出 WML 文档与标准 WML 语法保持一致所需的行。一个 xsl:template 定义了 catalog 元素 (第 16~46 行)。第 18 行是 wml 元素的开始。该元素称为“卡片组”, 因其包含一个或多个逻辑性的页面, 这些页称为“卡片”。所有卡片都用 card 元素标记 (第 25 行)。

常见编程错误 23.1 WML 元素名要区分大小写, 而且必须采用小写形式。WML 元素名包含大写字母是语法错误。这种错误会造成页面无法由微型浏览器正确显示。

第 20~23 行定义 head 元素。尽管 head 元素并非 WML 必需的, 但本例的 head 元素含有一个 meta 标记, 它禁止客户缓存当前页。由于带宽有限, 大多数 WML 客户都将下载的页缓存到内存中, 以备将来使用。但缓存以前访问过的页, 会造成 Web 客户不能总是显示网页的最新版本。第 21~22 行的 meta 标记则可禁止缓存, 确保客户接收每个页的最新版本。

第 25 行开始一个 card 元素, 并为其定义了 title 属性。该属性用于对卡片进行描述, 它的值肯定会由微型浏览器显示在屏幕顶部 (参见屏幕截图)。第 27 行开始一个 p 元素。在 WML 文档中, 文本段落标记 (<p>和</p>) 标记要在微型浏览器中显示的文本。WML 文档中的所有文本都必须放到<p>标记之间, 这些标记又嵌套在<card>标记内。第 33~38 行将 product 模板的任何匹配项插入文档。这些匹配项按照它们的 title 元素排序 (第 36 行), 从而确保书籍列表的标题名称按字母顺序排列。

```

1  <?xml version = "1.0"?>
2
3  <!-- Fig. 23.30: allBooks.xsl -->
4  <!-- XSLT style sheet that transforms XML generated by -->
5  <!-- books.py into WML. -->
6
7  <xsl:stylesheet version = "1.0"
8    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10   <xsl:output method = "xml" omit-xml-declaration = "no"
11     indent = "yes" doctype-system =
12       "http://www.wapforum.org/DTD/wml12.dtd"
13     doctype-public = "-//WAPFORUM//DTD WML 1.2//EN"/>
14
15   <!-- template for catalog element -->
16   <xsl:template match = "catalog">
17
18     <wml>
19
20       <head>
21         <meta forua = "true" http-equiv = "Cache-Control"
22           content = "max-age=0" />
23       </head>
24
25       <card title = "Book List">
26
27         <p>
28
29           <b>Available Books</b><br />
30           Click a link to view book information<br />
31
32           <!-- match product elements to product template -->
33           <xsl:apply-templates select = "/catalog/product">
34
35             <!-- sort products by title -->
36             <xsl:sort select = "title" />
37
38           </xsl:apply-templates>
39
40         </p>

```

```

41
42     </card>
43
44 </wml>
45
46 </xsl:template>
47
48 <!-- template for building row of Product information -->
49 <xsl:template match = "product">
50
51     <a href = "displayBook.py?ID=%s&isbn={isbn}">
52         <xsl:value-of select = "title" />, <xsl:value-of
53             select = "editionNumber" />e
54     </a><br />
55
56 </xsl:template>
57
58 </xsl:stylesheet>

```



图 23.30 用于 WML 客户的 allBooks.xsl

第 49~56 行为 product 元素定义一个 xsl:template。第 51 行指定一个 anchor 标记，并设置了 href 属性。href 属性的值引用 displayBook.py，并向其传递一个查询字符串，其中包含会话 ID 和 XML 文档的 isbn 元素的值。记住 allBooks.py 的第 71 行将会话 ID 插入这个查询字符串中。当前 product 的 isbn 使用 {isbn} 插入。最终的查询字符串使 displayBook 能识别客户并显示书籍。anchor 标记包含了文本，以及 XML 文档 title 及 editionNumber 元素的值（第 52~53 行）。

图 23.31 是用于 WML 客户的 displayBook.xsl。它将 displayBook.py（图 23.12）生成的 XML 转换成 WML，以便无线客户呈现。

第 27 行将书的 title 放到文档的 card 元素的 title 属性中，第 31 行将 title 放到位于文档的 p 元素开头的段落中。第 33 行指定一个 img 元素，src 值设为 "/bookstore/logo.wbmp"，而 alt 值设为 "logo"。该元素指定了要在书的标题下显示的图像名称。

注意对于 WML 客户，我们使用名为 logo.wbmp 的一幅图像，它位于根文件夹 bookstore 中。这是一幅无线位图图像。大多数微型浏览器都支持位图（.bmp）图像，但有的（比如 Openwave Simulator）只支持无线位图格式（wbmp）。位图是指用一系列二进制位表示的图像，它们对应于像素。像素对应于屏幕上的颜色点。可用许多图像编辑软件来创建这种图像，比如 Paint Shop Pro（www.jasc.com）或者

Photoshop Elements (www.adobe.com).

利用 Pic2WBMP (www.gingco.de/wap) 等转换程序可将普通位图转换成无线位图。另一个办法是扩展 Photoshop Elements 的功能, 使其支持.wbmp 格式。RCP Distributed Systems 公司就提供了一个插件, 可进行这样的功能扩展。下载和安装指南请从 www.rcp.co.uk/distributed/Downloads 免费下载。

第 37、第 38 行、第 39 行和第 41 行分别显示书的 price、isbn、editionNumber 和 copyright。第 45~46 行创建 Add to Cart (addToCart.py) 和 View Cart (viewCart.py) 这两个超链接。注意创建超链接而不是按钮, 目的是在 WML 客户上简化设计 (WML 表单请参见图 23.33)。

图 23.32 展示了用于 WML 客户的 viewCart.xsl, 它将 viewCart.py (图 23.16) 生成的 XML 转换成 WML, 以便由客户呈现。

```

1  <?xml version = "1.0"?>
2
3  <!-- Fig. 23.31: displayBook.xsl -->
4  <!-- XSLT style sheet that transforms XML generated by -->
5  <!-- displayBook.py into WML. -->
6
7  <xsl:stylesheet version = "1.0"
8      xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10     <xsl:output method = "xml" omit-xml-declaration = "no"
11         indent = "yes" doctype-system =
12             "http://www.wapforum.org/DTD/wml12.dtd"
13             doctype-public = "-//WAPFORUM/DTD WML 1.2//EN"/>
14
15     <!-- specify the root of the XML document -->
16     <!-- that references this style sheet -->
17     <xsl:template match = "product">
18
19         <wml>
20
21             <head>
22                 <meta forua = "true" http-equiv = "Cache-Control"
23                     content = "max-age=0" />
24             </head>
25
26             <!-- obtain book title from script to place in title -->
27             <card title = "{title}">
28
29                 <p>
30
31                     <xsl:value-of select = "title" /><br />
32
33                     <img src = "/bookstore/logo.wbmp" alt = "logo" />
34                     <br />
35
36                     <!-- show price, ISBN, edition and copyright -->
37                     Price: <xsl:value-of select = "price" /><br />
38                     ISBN: <xsl:value-of select = "isbn" /><br />
39                     Edition: <xsl:value-of select = "editionNumber" />
40                     <br />
41                     Copyright: <xsl:value-of select = "copyright" /><br />
42                     <br />
43
44                     <!-- add links to shopping cart -->
45                     <a href = "addToCart.py?ID=%s">Add to Cart</a><br />
46                     <a href = "viewCart.py?ID=%s">View Cart</a><br />
47
48                 </p>
49
50             </card>
51
52         </wml>
53
54     </xsl:template>
55
56 </xsl:stylesheet>

```



图 23.31 用于 WML 客户的 displayBook.xsl

```

1 <?xml version = "1.0"?>
2
3 <!-- Fig. 23.32: viewCart.xsl -->
4 <!-- XSLT style sheet that transforms XML generated by -->
5 <!-- viewCart.py into WML. -->
6
7 <xsl:stylesheet version = "1.0"
8   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10   <xsl:output method = "xml" omit-xml-declaration = "no"
11     indent = "yes" doctype-system =
12       "http://www.wapforum.org/DTD/wml12.dtd"
13     doctype-public = "-//WAPFORUM//DTD WML 1.2//EN"/>
14
15   <xsl:template match = "cart">
16
17     <wml>
18
19       <head>
20         <meta forua = "true" http-equiv = "Cache-Control"
21           content = "max-age=0" />
22       </head>
23
24       <card title = "Shopping Cart">
25
26         <p>
27
28           <b>Shopping Cart</b><br /><br />
29
30           <xsl:choose>
31             <xsl:when test = "@total = '0.00'">
32               Your shopping cart is currently empty.<br />
33             <br />
34             </xsl:when>
35
36             <xsl:otherwise> <!-- total != 0.00 -->
37

```

```

38         <xsl:apply-templates select = "orderProduct">
39
40             <!-- sort orderProducts by title -->
41             <xsl:sort select = "product/title" />
42
43         </xsl:apply-templates>
44
45         <b>Total: <xsl:value-of
46             select = "@total" /></b><br />
47
48         </xsl:otherwise>
49     </xsl:choose>
50
51     <a href = "allBooks.py?ID=%s">Continue Shopping</a>
52     <br />
53     <a href = "order.py?ID=%s">Check Out</a>
54
55 </p>
56 </card>
57
58 </wml>
59
60 </xsl:template>
61
62 <xsl:template match = "orderProduct">
63
64     Product:<br />
65     <xsl:value-of select = "product/title" />,
66     <xsl:value-of select = "product/editionNumber" /><br />
67     Quantity: <xsl:value-of select = "quantity" /><br />
68     Price: <xsl:value-of select = "product/price" /><br />
69     Subtotal: <xsl:value-of select = "subtotal" /><br />
70     <br />
71
72 </xsl:template>
73
74 </xsl:stylesheet>

```



图 23.32 用于 WML 客户的 viewCart.xsl

第一个 xsl:template (第 15~61 行) 匹配 cart 元素。第 30 行开始一个 xsl:choose 元素。如果 cart 的 total 属性 (用 @ 字符指示) 等于 "0.00", 程序向客户显示一条消息, 指出购物车目前是空的 (第 32~33 行); 否则, 就显示购物车中的所有商品 (第 36~48 行)。

第 38~43 行将所有匹配项插入 orderProduct 模板, 并按它们的 product/title 元素进行排序。orderProduct 模板 (第 63~73 行) 匹配 orderProduct 元素。第 66~70 行在单独的行上显示 orderProduct 的 product/title、product/editionNumber、quantity、product/price 和 subtotal。接着, 第 45~46 行显示全部商品总计金额。第 51~53 行为用户创建两个超链接选项, 分别指向 allBooks.py 和 order.py。

图 23.33 是用于 WML 客户的 orderForm.wml, 是 order.py (图 23.18) 向 WML 客户显示的订单表单。

```

1  <!-- Fig. 23.33: orderForm.wml -->
2  <!-- Static WML to be displayed by order.py -->
3
4  <!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.2//EN"
5    "http://www.wapforum.org/DTD/wml12.dtd">
6
7  <wml>
8
9    <head>
10      <meta forua = "true" http-equiv = "Cache-Control"
11        content = "max-age=0"/>
12    </head>
13
14    <card title = "Order">
15
16      <p>
17
18        <b>Shopping Cart Check Out</b><br /><br />
19
20        <!-- form to input user information and credit card -->
21        <!-- note: no need to input real data in this example -->
22        Input the following:<br />
23        First name: <input type = "text" name = "firstname" />
24        <br />
25        Last name: <input type = "text" name = "lastname" />
26        <br />
27        Street: <input type = "text" name = "street" /><br />
28        City: <input type = "text" name = "city" /><br />
29        State: <input type = "text" name = "state" /><br />
30        Zip code: <input type = "text" name = "zipcode" /><br />
31        Phone #: <input type = "text" name = "phone" /><br />
32        Credit Card #: <input type = "text"
33          name = "creditcard" /><br />
34        Expiration (mm): <input type = "text" name = "expires" />
35        Expiration (yyyy): <input type = "text"
36          name = "expires2" /><br />
37
38        <do type = "accept" label = "Submit">
39
40          <go method = "post" href = "process.py?ID=%s">
41            <postfield name = "firstname"
42              value = "${firstname}" />
43            <postfield name = "lastname"
44              value = "${lastname}" />
45            <postfield name = "street" value = "${street}" />
46            <postfield name = "city" value = "${city}" />
47            <postfield name = "state" value = "${state}" />
48            <postfield name = "zipcode" value = "${zipcode}" />
49            <postfield name = "phone" value = "${phone}" />
50            <postfield name = "creditcard"
51              value = "${creditcard}" />
52            <postfield name = "expires"
53              value = "${expires}" />
54            <postfield name = "expires2"
55              value = "${expires2}" />
56          </go>
57

```

```

58         </do>
59
60     </p>
61
62 </card>
63
64 </wml>

```



图 23.33 用于 WML 客户的 orderForm.wml

第 23~36 行创建 input 元素以便从用户获得信息。input 元素在屏幕上创建一个“文本字段”，以便用户通过无线设备的小键盘输入信息。第 38~58 行使用了 do 和 go 元素。其中，do 元素可创建无线设备的“软键”。软键是无线设备上的物理按钮，利用它们可在不同文档间切换。在屏幕中，do 元素的 label 属性值会在相应的按钮上显示。按下软键，就能执行由 go 元素指定的动作。

do 元素的一个属性是 type。type 属性最常用的两个值是“accept”和“options”。前者对应于左软键，而后者对应于右软键。

第 38~58 行使用 do 和 go 元素创建一个软键链接。type 属性值是“accept”，所以当用户向下滚动到表单底部之后，会在显示屏左下角显示“Submit”字样（即 label 属性的值），指出左软键当前的功能是什么。按下软键，就会执行由 go 元素指定的动作，即将用户重定向到 process.py。

go 元素类似于 a 元素，都用于链接到特定的资源。go 元素的 href 属性值定义了链接地址。在图 23.33 中，go 元素链接到 process.py，使用 post 表单方法，并在查询字符串中指定会话 ID（第 40 行）。

第 41~55 行使用 postfield 元素将数据插入 go 元素。postfield 元素有两个属性是必需的，即 name 和 value。和 XHTML 的 input 元素一样，postfield 的 name 属性指定用于引用数据的名称，而 value 属性指定实际数据。在图 23.33 中，每个 value 属性的值都从对应的 input 元素取得。例如在第 44 行，value 属性的值是“\${lastname}”，表明传递的值要从 name 属性为 lastname 的 input 字段取得（第 25 行）。数据要传给由 go 元素的 href 属性指定的目的地。客户提交表单后，就会取得数据值，而 go 元素将用户重定向到 process.py。

图 23.34 是用于 WML 客户的 thankYou.html，它由 process.py（图 23.20）显示给 WML 客户。图 23.35

是用于 WML 客户的 error.xsl，它将 error.py（图 23.22）生成的 XML 转换成 WML，以便由客户呈现。第 15~37 行定义一个 xsl:template 以匹配 error 元素。第 29 行将 message 元素值插入 paragraph 标记（第 26 行）。

```

1  <!-- Fig. 23.34: thankYou.wml          -->
2  <!-- Static WML to be displayed by process.py  -->
3
4  <!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.2//EN"
5    "http://www.wapforum.org/DTD/wml12.dtd">
6
7  <wml>
8
9    <head>
10      <meta forua = "true" http-equiv = "Cache-Control"
11        content = "max-age=0" />
12    </head>
13
14    <card title = "Thank You!">
15
16      <p>
17
18        <b>Thank You</b><br /><br />
19        Your order has been processed.<br />
20        Your credit card has been billed: <b>%.2f</b>
21
22      </p>
23
24    </card>
25
26  </wml>

```



图 23.34 用于 WML 客户的 thankYou.wml

```

1  <?xml version = "1.0"?>
2
3  <!-- Fig. 23.35: error.xsl          -->
4  <!-- XSLT style sheet that transforms XML generated by  -->
5  <!-- error.py into WML.          -->
6
7  <xsl:stylesheet version = "1.0"

```

```

8  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
9
10 <xsl:output method = "xml" omit-xml-declaration = "no"
11     indent = "yes" doctype-system =
12     "http://www.wapforum.org/DTD/wml12.dtd"
13     doctype-public = "-//WAPFORUM//DTD WML 1.2//EN"/>
14
15 <xsl:template match = "error">
16
17     <wml>
18
19         <head>
20             <meta forua = "true" http-equiv = "Cache-Control"
21                 content = "max-age=0" />
22         </head>
23
24         <card title = "Error">
25
26             <p>
27
28                 <b>Error message:</b><br />
29                 <xsl:value-of select = "message" />
30
31             </p>
32
33         </card>
34
35     </wml>
36
37 </xsl:template>
38
39 </xsl:stylesheet>

```



图 23.35 用于 WML 客户的 error.xsl

23.15 因特网和万维网资源

以下网址提供了和本章讨论的主题有关的更多的信息。

www.wapforum.org

WAP 论坛负责制定 WAP 建议规范以建立无线设备互操作性，本网站介绍 WAP 历史及其现状。

www.w3.org/TR/xhtml-basic

在 XHTML Basic 建议规范中，包含 XHTML Basic 的所有细节和要素。

www.zvon.org/xxl/xhtmlBasicReference/Output/index.html

这是一个 XHTML Basic 参考网站，提供大量示例。

www.softsteel.co.uk/tutorials/wmltut/

提供简短的 WML 教程。

www.webtools.com/story/html/TLS20000818S0001

简介 WML，并提供相应的教程。

第 24 章 多媒体

学习目标

- 介绍 Python 多媒体应用程序编程
- 学会使用 PyOpenGL 模块创建三维对象
- 用 Python 和 Alice 操纵三维对象
- 学会使用 Python 和 pygame 创建多媒体应用程序

24.1 概述

10 年前的桌面计算机功能有限,不足以在应用程序中集成高质量声音和视频。但自那时起,计算机技术的飞速发展使多媒体迅速普及,用户可以使用丰富的 CD-ROM 和 DVD 节目,并可在网上享受流式声音/视频内容。本章概述如何使用 3 种技术(PyOpenGL、Alice 和 pygame),利用 Python 的多媒体能力来创建交互式应用程序。

24.2 PyOpenGL 简介

Python 的 PyOpenGL 模块封装了“OpenGL 应用程序编程接口”(OpenGL API),后者是用于呈现二维(2D)和三维(3D)图形的一个函数库。程序员使用 PyOpenGL 模块在 Python 程序中集成 2D 和 3D 图形。虽然本章并不讲解 OpenGL 本身,但掌握了 PyOpenGL 的知识后,即使以前没有使用 OpenGL 的经验,也能顺利编写出色的应用程序。

“OpenGL Utility Toolkit”(GLUT)、wxPython、Tkinter 和 FxPy 都是能提供 OpenGL 背景(即一个电子画布)的 API,OpenGL 呈现的图形会在这个背景上显示。本章的例子将 Tkinter 作为 OpenGL 的背景使用。

PyOpenGL 模块包含名为 Opengl 的 Tkinter 组件,它能显示 OpenGL 图形,也允许程序员使用 Tkinter 组件和布局管理器。除此之外,Opengl 组件已为每个鼠标按钮绑定了一个事件。该组件允许用户按住鼠标左键来移动对象。要想旋转对象,可按住鼠标中键(如果有的话)。要改变对象大小,可按住鼠标右键并拖动。

PyOpenGL 模板提供了 Windows 和 Linux/UNIX 系统的版本,下载地址是 sourceforge.net/projects/pyopengl。本章的例子使用 PyOpenGL v2.0.0.44。它的安装指南可参见 Deitel 公司网站(www.deitel.com)。

24.3 PyOpenGL 示例

本节展示两个创建三维形状的 PyOpenGL 示例。图 24.1 使用 PyOpenGL 着色和变形功能创建一个旋转彩盒。注意本章 PyOpenGL 示例的程序结构类似于第 10 章和第 11 章的 GUI 程序,因为这些示例使用 Tkinter(Python 的标准 GUI 工具)来创建窗口,以便在其中显示 PyOpenGL 图形。

第 83 行创建 ColorBox 类(第 8~80 行)的一个对象,并调用它的 mainloop 方法来启动 GUI。ColorBox 构造函数(第 11~28 行)创建并初始化 Tkinter 窗口(第 14~17 行)。第 20~21 行创建并 pack 一个名为 OpenGL 的 Opengl 组件,它负责呈现 OpenGL 对象。Opengl 组件的 double 属性设为 1,以使用双缓冲。双缓冲要维护两个屏幕缓冲区,一个用于显示(屏上缓冲区),另一个用于绘图(屏下缓冲区)。应用程序在“屏下缓冲区”描绘下一帧(即要显示的下一幅图)。程序结束下一帧的绘图后,会交换两个缓

缓冲区，以显示下一帧内容。相反，如使用的是单缓冲，程序只能直接在屏幕上绘图，造成显示不是很流畅，因为程序要在描绘每个新帧之前，先清除绘图区域。取决于计算机系统的速度，清除操作可能造成在显示下一帧之前，背景色出现短暂的闪烁。

```

1 # Fig 24.1: fig24_01.py
2 # Colored, rotating box (with open top and bottom).
3
4 from Tkinter import *
5 from OpenGL.GL import *
6 from OpenGL.Tk import *
7
8 class ColorBox( Frame ):
9     """A colorful, rotating box"""
10
11     def __init__( self ):
12         """Initialize GUI and OpenGL"""
13
14         Frame.__init__( self )
15         self.master.title( "Color Box" )
16         self.master.geometry( "300x300" )
17         self.pack( expand = YES, fill = BOTH )
18
19         # create and pack OpenGL -- use double buffering
20         self.openGL = OpenGL( self, double = 1 )
21         self.openGL.pack( expand = YES, fill = BOTH )
22
23         self.openGL.redraw = self.redraw # set redraw function
24         self.openGL.set_eyepoint( 20 ) # move away from object
25
26         self.amountRotated = 0 # total degrees of rotation
27         self.increment = 2 # rotate amount in degrees
28         self.update() # begin rotation
29
30     def redraw( self, openGL ):
31         """Draw box on white background"""
32
33         # clear background and disable lighting
34         glClearColor( 1.0, 1.0, 1.0, 0.0 ) # set clearing color
35         glClear( GL_COLOR_BUFFER_BIT ) # clear background
36         glDisable( GL_LIGHTING )
37
38         # constants
39         red = ( 1.0, 0.0, 0.0 )
40         green = ( 0.0, 1.0, 0.0 )
41         blue = ( 0.0, 0.0, 1.0 )
42         purple = ( 1.0, 0.0, 1.0 )
43
44         vertices = \
45             [ ( ( -3.0, 3.0, -3.0 ), red ),
46               ( ( -3.0, -3.0, -3.0 ), green ),
47               ( ( 3.0, 3.0, -3.0 ), blue ),
48               ( ( 3.0, -3.0, -3.0 ), purple ),
49               ( ( 3.0, 3.0, 3.0 ), red ),
50               ( ( 3.0, -3.0, 3.0 ), green ),
51               ( ( -3.0, 3.0, 3.0 ), blue ),
52               ( ( -3.0, -3.0, 3.0 ), purple ),
53               ( ( -3.0, 3.0, -3.0 ), red ),
54               ( ( -3.0, -3.0, -3.0 ), green ) ]
55
56         glBegin( GL_QUAD_STRIP ) # begin drawing
57
58         # change color and plot point for each vertex
59         for vertex in vertices:
60             location, color = vertex
61             apply( glColor3f, color )
62             apply( glVertex3f, location )
63
64         glEnd() # stop drawing
65

```

```

66 def update( self ):
67     """Rotate box"""
68
69     if self.amountRotated >= 500: # change rotation direction
70         self.increment = -2      # rotate left
71     elif self.amountRotated <= 0: # change rotation direction
72         self.increment = 2       # rotate right
73
74     # rotate box around x, y, z axis ( 1.0, 1.0, 1.0 )
75     glRotate( self.increment, 1.0, 1.0, 1.0 )
76     self.amountRotated += self.increment
77
78     self.openGL.tkRedraw()        # redraw geometry
79     self.openGL.after( 10, self.update ) # call update in 10ms
80
81 def main():
82     ColorBox().mainloop()
83
84 if __name__ == "__main__":
85     main()

```

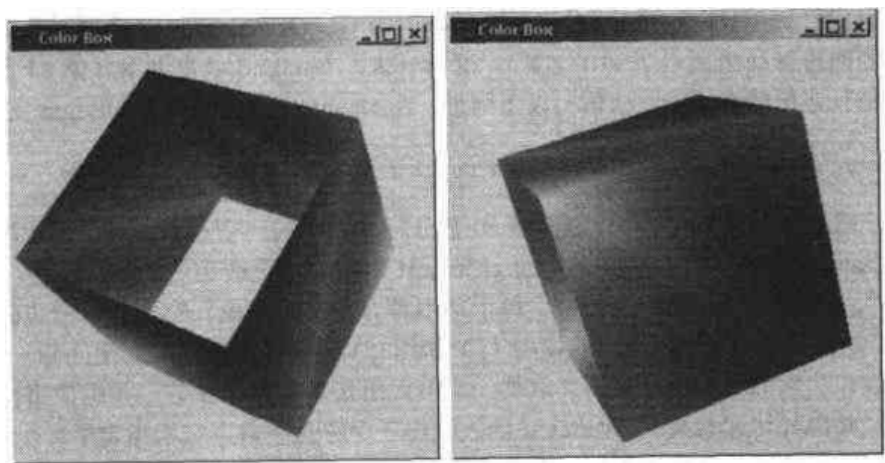


图 24.1 在 Tkinter 背景上使用 PyOpenGL

第 23 行设置 `opengl` 的 `redraw` 方法，每次图像发生改变时，程序调用它以更新屏上显示。第 24 行调用 PyOpenGL 的 `set_eyepoint`（设置视点）方法，使照相机移离布景（即三维图像）一个由程序员指定的量。照相机定义了观察者相对于布景的视角。向 `set_eyepoint` 传递较大的值，会造成屏幕上的图像变小，因为视点远离图像。设为负值，可从背后查看图像。

第 26~27 行初始化变量 `amountRotated` 和 `increment`。这些值控制盒子的旋转度数。在这个程序中，每次旋转图像，都会度数增加 2。旋转的总度数存储在变量 `amountRotated` 中。图像总共旋转 500 度，就改变旋转方向。最后，第 28 行调用 `update` 方法（在第 66~79 行定义）来旋转图像，重画图像，并指出程序应在 10 毫秒后再次调用 `update`。

`redraw` 方法（第 30~65 行）将背景设为白色，并描绘盒子。第 34 行调用 PyOpenGL 的 `glClearColor` 函数来指定 `glClear` 函数使用的颜色。通常，可用一个三元素元组来表示颜色，元素值的范围在 0.0~1.0 之间，分别表示一种颜色的红、绿和蓝成份。也可用一个四元素元组来表示颜色，它的前三个元素代表红、绿、蓝成份，最后一个元素则代表 `alpha` 值（或透明度）。颜色的 `alpha` 值决定该颜色在一幅图中呈现时，如何与其他颜色合并。颜色要么是纯色（`alpha` 值为 0.0），要么完全透明（`alpha` 值为 1.0），要么在这两种极端情况之间。0.0~1.0 之间的 `alpha` 值决定了透明度。`glClearColor` 要求具有红、绿、蓝和 `alpha` 值的一种颜色作为参数。组合不同的值，就可获得不同的颜色。例如，纯白是 (1.0, 1.0, 1.0, 0.0)，而纯黑是 (0.0, 0.0, 0.0, 0.0)。

第 35 行调用 PyOpenGL `glClear` 函数来呈现白色背景。传给 `glClear` 的值是 `GL_COLOR_BUFFER_BIT`，它指出函数应清除“颜色缓冲区”。颜色缓冲区存储与窗口的每个像素有关

的信息，每个像素都对应屏幕上显示的一个颜色点。缓冲区存储的是每个像素的红、绿和蓝颜色成份。PyOpenGL 的 `glClear` 函数将 `glClearColor` 函数指定的颜色存储到颜色缓冲区的每个条目中。这样就可替换缓冲区中的任何旧值。这实际会清除 `glClearColor` 函数以前指定的颜色，将背景设为白色。第 39~42 行定义盒子采用的其他颜色。

PyOpenGL 的 `glDisable` 函数（第 36 行）禁止“照明”（由 `GL_LIGHTING` 参数指定）。计算机图形的照明是指在布景上放置光源。OpenGL 有 10 个参数可指定光源（例如位置、方向、灯光颜色等等）。如启用照明，对象颜色会根据光源参数来计算；否则，就将 `glClearColor` 指定的颜色应用于光源。

程序接着创建 3D 盒子。为了用 OpenGL 绘图，程序员需要指定“顶点”。顶点要放在 `glBegin` 和 `glEnd` 函数调用之间。传给 `glBegin` 的参数定义要描绘什么类型的形状。

第 44~54 行创建定义盒子形状的顶点列表。列表中每个元组都包含一个顶点位置和一个颜色。第 56~64 行描绘盒子。第 56 行调用 PyOpenGL 的 `glBegin` 函数，并传递参数 `GL_QUAD_STRIP`，它要求 OpenGL 描绘一组相互连接的四边形来构成一个四面体。程序使用列表中的前四个顶点（顶点 1~4）描绘第一个四边形，第二个四边形由列表中的顶点 3~6 指定，以此类推。注意第 44~54 行定义的列表有 10 个顶点，而且最后一对顶点与第一对相同，这是因为 `GL_QUAD_STRIP` 需要最后两个顶点来描绘连接成一个盒子的四边形（由顶点 7~10 定义）。这样一来，在 `glEnd` 函数调用（第 64 行）之前，一系列顶点就定义了相互连接的多边形。请访问以下网址，在 OpenGL 文档中查看 `glBegin` 支持的其他参数：

www.opengl.org/developers/documentation/man_pages/hardcopy/GL/html/gl/begin.htm

为描绘盒子，需要将顶点指定为 3D 点。第 60 行分解列表中的一个元组，获得每个顶点的位置及颜色。第 61 行使用 `apply` 函数调用 PyOpenGL 的 `glColor3f` 函数，以更改当前的绘图颜色。`glColor3f` 函数取得代表 RGB 颜色的三个浮点数作为参数。每个顶点都有自己的颜色，它们在一个边的中央会聚时，PyOpenGL 会混合这些颜色。`apply` 函数（第 62 行）调用 `glVertex3f` 函数在三维空间画一个点。`apply` 函数取得另一个函数作为它的第一个参数，并取得一个值元组作为第二个参数，元组值将传给那个函数。方法将调用函数，并向其传递参数。`glVertex3f` 函数取得三个浮点参数，它们指定了一个点的位置。

这个例子还展示了如何为三维图形添加动画。`update` 方法（第 66~79 行）使盒子旋转。第 70~73 行改变由 `increment` 变量所表示的旋转方向。`increment` 为正值，盒子将逆时针旋转，负值则使盒子顺时针旋转。盒子在一个方向旋转 500 度之后，程序就改变旋转方向（第 70~73 行）。`glRotate` 方法（第 76 行）接受四个参数。第一个参数是 `increment` 变量，以度数为单位设置旋转角度。另外三个浮点数描述了旋转中心线。这条线穿过原点 (0.0, 0.0, 0.0) 和指定点 (1.0, 0.0, 0.0)。第 77 行使 `amountRotated` 变量自增，该变量指定了盒子的旋转度数。调用 `tkRedraw` 方法（第 79 行），就可用旋转好的形状来更新 OpenGL 组件。`after` 方法（第 80 行）取得两个参数——值 10 和 `update` 方法。结果就是程序的事件循环 `mainloop` 每隔 10 毫秒调用一次 `update`。

图 24.2 演示了如何使用 OpenGL.GLUT 模块的几个方法来创建三维形状（GLUT 是 GL Utilities Toolkit 的简称）。这个例子创建一个 GUI，以便预览对象的颜色和形状。

```
1 # Fig. 24.2: fig24_02.py
2 # Demonstrating various GLUT shapes.
3
4 from Tkinter import *
5 import Prw
6 from OpenGL.GL import *
7 from OpenGL.Tk import *
8 from OpenGL.GLUT import *
9
10 class ChooseShape( Frame ):
11     """Allow user to preview different shapes and colors"""
12
13     def __init__( self ):
14         """initialize GUI and OpenGL"""
15
16         Frame.__init__( self )
```

```

17 Pmw.initialise()
18 self.master.title( "Choose a shape and color" )
19 self.master.geometry( "300x300" )
20 self.pack( expand = YES, fill = BOTH )
21
22 # create and pack MenuBar
23 self.choices = Pmw.MenuBar( self )
24 self.choices.pack( fill = X )
25
26 # create and pack OpenGL -- use double buffering
27 self.openGL = OpenGL( self, double = 1 )
28 self.openGL.pack( expand = YES, fill = BOTH )
29
30 self.openGL.redraw = self.redraw # set redraw function
31 self.openGL.set_eyepoint( 20 ) # move away from object
32 self.openGL.autospin_allowed = 1 # allow auto-spin
33
34 self.choices.addmenu( "Shape", None ) # Shape submenu
35
36 # possible shapes and arguments
37 self.shapes = { "glutWireCube" : ( 3, ),
38                 "glutSolidCube": ( 3, ),
39                 "glutWireIcosahedron" : {},
40                 "glutSolidIcosahedron" : {},
41                 "glutWireCone" : ( 3, 3, 50, 50 ),
42                 "glutSolidCone" : ( 3, 3, 50, 50 ),
43                 "glutWireTorus" : ( 1, 3, 50, 50 ),
44                 "glutSolidTorus" : ( 1, 3, 50, 50 ),
45                 "glutWireTeapot" : ( 3, ),
46                 "glutSolidTeapot" : ( 3, ) }
47
48 self.selectedShape = StringVar()
49 self.selectedShape.set( "glutWireCube" )
50
51 sortedShapes = self.shapes.keys()
52 sortedShapes.sort() # sort names before adding to menu
53
54 # add items to Shape menu
55 for shape in sortedShapes:
56     self.choices.addmenuitem( "Shape", "radiobutton",
57                               label = shape, variable = self.selectedShape,
58                               command = self.refresh )
59
60 self.choices.addmenu( "Color", None ) # Color submenu
61
62 # possible colors and their values
63 self.colors = { "Black" : ( 0.0, 0.0, 0.0 ),
64                 "Blue" : ( 0.0, 0.0, 1.0 ),
65                 "Red" : ( 1.0, 0.0, 0.0 ),
66                 "Green" : ( 0.0, 1.0, 0.0 ),
67                 "Magenta" : ( 1.0, 0.0, 1.0 ) }
68
69 self.selectedColor = StringVar()
70 self.selectedColor.set( "Black" )
71
72 # add items to Color menu
73 for color in self.colors.keys():
74     self.choices.addmenuitem( "Color", "radiobutton",
75                               label = color, variable = self.selectedColor,
76                               command = self.refresh )
77
78 def redraw( self, openGL ):
79     """Draw selected shape on white background"""
80
81     # clear background and disable lighting
82     glClearColor( 1.0, 1.0, 1.0, 0.0 ) # select clearing color
83     glClear( GL_COLOR_BUFFER_BIT ) # clear background
84     glDisable( GL_LIGHTING )
85
86     # obtain and set selected color
87     color = self.selectedColor.get()

```

```

88     apply( glColor3f, self.colors[ color ] )
89
90     # obtain and draw selected shape
91     shape = self.selectedShape.get()
92     apply( eval( shape ), self.shapes[ shape ] )
93
94     def refresh( self ):
95         self.openGL.tkRedraw()
96
97 def main():
98     ChooseShape().mainloop()
99
100 if __name__ == "__main__":
101     main()

```

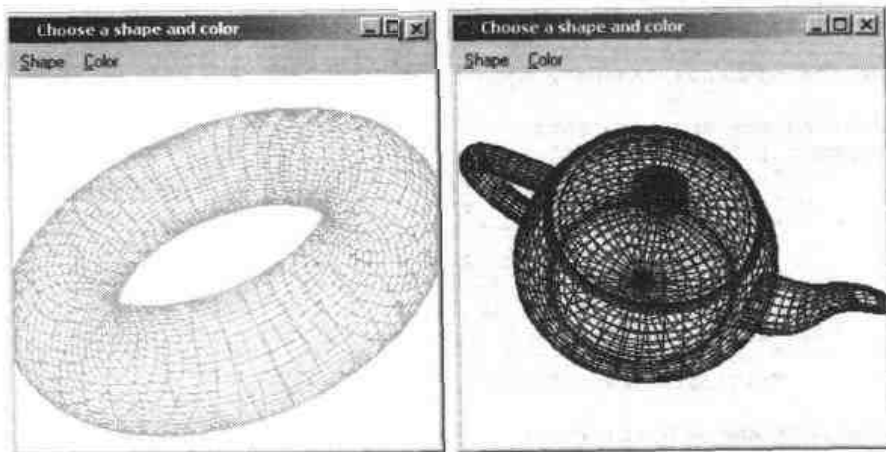


图 24.2 用 GLUT 创建各种形状

第 98 行创建 `ChooseShape` 类（第 10~95 行）的一个对象，并调用 `mainloop` 来开始应用程序的事件循环。第 16~20 行初始化 GUI 窗口。第 23~24 行创建并 `pack` 一个 `MenuBar` 组件。第 27~32 行初始化 `OpenGL` 组件。第 32 行将 `autospin_allowed` 设为 1，它允许用户按住鼠标中键（要求三键鼠标），并朝希望的方向拖动鼠标，从而连续旋转一个形状。

本例为用户提供了一个菜单，以便选择要预览的形状，包括立方体、二十面体、圆锥体、圆环体（就像一个面包圈）以及茶壶。可选择查看骨架结构和实心体。图 24.2 的屏幕截图显示了骨架结构的圆环体和茶壶。

`shapes` 字典（第 37~46 行）将 GLUT 形状作为它的键。每个键都对应一个用来描绘 3D 形状的函数名。值则可能是传给那些函数的参数。程序还允许用户选择形状的颜色。`colors` 字典（第 63~67 行）用一系列颜色名称作为键。每个颜色都用它的 RGB 元组作为它的值。GUI 提供一个 `Pmw MenuBar`，并为每种形状和颜色设置了一个单选钮菜单项，默认选择是白色骨架结构立方体。第 73~76 行为 `Color` 菜单添加菜单项。从 `Color` 菜单选择一项，会调用 `refresh` 方法（第 94~95 行）。

选择菜单项会更新显示。首先，`refresh` 方法调用 `tkRedraw` 来更新窗口。`tkRedraw` 函数调用与窗口对应的 `redraw` 函数。`redraw` 函数（第 78~92 行）将背景设为白色，并禁止照明。第 87 行获得所选的颜色。`apply` 方法（第 89 行）将颜色的 RGB 值传给函数 `glColor3f`，后者设置绘图颜色。类似地，第 91 行获得所选的形状。`eval` 函数将代表形状函数的一个字符串转换成真正的函数调用，`apply` 调用指定的函数，并传递与形状的字典键对应的任何参数。

GLUT 库提供了描绘三维形状的函数，它们可在本例中使用。`glutWireCube` 和 `glutSolidCube` 函数接收立方体的边长作为参数，本例是 3。`glutWireIcosahedron` 和 `glutSolidIcosahedron` 函数则没有参数，它们直接创建半径为 1.0 的二十面体。

`glutWireCone` 和 `glutSolidCone` 函数取得四个参数。前两个参数对应于基面半径和圆锥高度。下一个参数是切片数量（即由尖端到基面的线条所定义的圆锥子区域数量）。如果圆锥只由三片构成，那么实际

是一个像金字塔那样的棱锥。增大切片数量,使“圆锥”越来越像一个真正的圆锥。第四个参数设置叠片数量(即平行于基面的横切片的数量)。

`glutWireTorus` 和 `glutSolidTorus` 函数也取得四个参数。前两个指定圆环体的内圈和外圈半径。内圈半径是内部挖空的半径(就像面包圈中间的洞),而外圈半径是指从中心点到最外圈的半径,它是整个圆环体的总半径。第三个参数指定了围绕整个圆环体的线条数。最后一个参数指定将圆环体的切片数量。`glutWireTeapot` 和 `glutSolidTeapot` 函数描绘一个茶壶,它的参数是该形状的相对大小。

24.4 Alice 简介

Alice (www.alice.org) 是一个 3D 交互式图形编程环境,使刚入门的 Python 程序员也能轻松进行 3D 建模。Alice 适用于 Microsoft Windows 操作系统,并提供了脚本编程环境,以便控制对象行为(即控制动画)。除了 Alice 自带的对象之外,还可在其中导入其他许多常用的 3D 建模格式。

程序员可用 Python 来控制 Alice 环境。Alice 提供一个简单、直观的界面,以便开发者将对象放到 Alice 的“世界”。世界是一个虚拟的 3D 布景。程序员设计好最初的布景后,一个 Python 脚本可使 Alice 世界中的对象“动”起来。Python 程序员还可访问每个对象的动作列表来创建动画。要在 Internet Explorer 中查看一个完成的世界,请从 www.alice.org/downloads/plugin 下载插件。

Alice 使用 Python 解释器的一个版本来解析 Python 代码。虽然 Python 通常要区分大小写,但 Alice 解释器不认为 Python 是一种大小写敏感的语言。Python 2.2 以及早期版本的用户还希望整数除法肯定会生成一个整数。但在 Alice 中,除法可能生成一个浮点数;例如,1/2 的结果是 0.5,而不是标准 Python 解释器所生成的 0。

24.5 狐狸、鸡和种子问题

我们将“狐狸、鸡和种子”问题作为一个游戏来实现,演示如何进行 Alice 编程(图 24.3)。游戏规则很简单: Alice Liddell 需要用一艘船将一只狐狸、一只鸡和一粒种子运过河。船很小,每次只能搭载一位乘客。但是,如果把狐狸和鸡留在同一个河岸,狐狸会吃鸡;同样地,鸡和种子也不能单独在一起。

要运行这个例子,必须安装 Alice Authoring Tool (Alice 创作工具)。详细安装指南可参见 www.deitel.com。要想运行这个例子,请启动 Alice Authoring Tool,打开“FoxChickenAndSeed”世界(从 www.deitel.com 下载)。选择 Animations 卡片,单击绿色的 Start 按钮以开始动画。例如,要想把狐狸移到船上,选择 fox 单选钮,再单击 Get into the boat 按钮。

图 24.3 展示了游戏控制面板和用于创建游戏的 Python 代码。注意本例使用 Alice 自带的一个花盆对象表示种子。本书没有提供用于呈现布景的 Alice 代码,请从 www.deitel.com 下载本书所有示例的压缩包,并找到其中的 `World_scene.py`。另外,本书也没有展示 Alice 世界的屏幕截图,因为印刷颜色有限,不足以准确呈现这个世界。

```
1  ### We omit the code generated by Alice. ###
2  # Fig. 24.3: fig24_03.py
3  # Fox, Chicken and Seed problem.
4
5  FollowTheBoat = Loop( camera.PointAt
6      ( AliceLiddell.dress.rthigh ) )
7
8  # run two animations together with given pause time
9  def AnimateWithPause( Animation1, Animation2, Object, time ):
10
11      return Loop( DoInOrder( DoTogether( Animation1, Animation2 ),
12          Object.Move( Forward, 0, Duration = time ) ) )
13
14  # create two fish following continuous animation pattern
15  LoopingFish = Loop( AnimateWithPause
```

```

16     ( Fish.Move( Forward, 50, Duration = 5 ),
17       Fish.Turn( Down, 1, Duration = 5 ), Fish, 15 ) )
18
19 LoopingFish2 = Loop( AnimateWithPause
20   { Fish2.Move( Forward, 70, Duration = 8 ),
21     Fish2.Turn( Down, 1, Duration = 8 ), Fish2, 25 ) )
22
23 # lists that keep track of object position
24 thisBank = [ "Fox", "Chicken", "Flower" ]
25 theBoat = []
26 otherBank = []
27
28 currentBank = thisBank
29 targetBank = otherBank
30 selected = None
31
32 # animal select callback
33 def animalSelect( value ):
34
35     global selected
36     selected = value
37
38 # put object into boat
39 def ObjectInBoat( Object ):
40
41     Object.RespondToCollisionWith( FishBoat.deck, Object.Stop )
42     Object.MoveTo( FishBoat.period )
43     Object.Move( Down, 2, Duration = 3 )
44
45 # get object out of boat
46 def ObjectOutOfBoat( Object ):
47
48     Object.RespondToCollisionWith( Ground, Object.Stop )
49     Object.Move( Left, 1 - int( ( len( Object._name ) - 1 ) / 3 ) )
50     Object.Move( Back, 7 )
51     Object.Move( Down, 3, Duration = 3 )
52
53 # put the currently selected object into boat
54 def getIntoBoat():
55
56     if selected in currentBank and
57       ( len( theBoat ) == 0 ) and boatArrived():
58         currentBank.remove( selected )
59         theBoat.append( selected )
60         ObjectInBoat( eval( selected ) )
61
62 # remove currently selected object from boat
63 def getOutOfBoat():
64
65     if selected in theBoat and boatArrived():
66         theBoat.remove( selected )
67         currentBank.append( selected )
68         ObjectOutOfBoat( eval( selected ) )
69
70 # send boat to other shore
71 def toOtherShore():
72
73     if not boatArrived(): # boat is still in transit
74         return
75
76     global currentBank, thisBank, otherBank
77
78     if len( theBoat ) == 1: # someone is on boat
79         DoInOrder( eval( theBoat[ 0 ] ).Move( Forward, 16,
80           Duration = 3 ), eval( theBoat[ 0 ] ).Turn( Left, 1/2, 1,
81           AsSeenby = FishBoat ) )
82
83     # move boat then set alarm to check rules
84     DoInOrder( FishBoat.Move( Forward, 16, Duration = 3 ),
85       FishBoat.Turn( Left, 1/2 ) )
86     Alice.SetAlarm( 1, checkRules, ( currentBank ) )

```



```

87
88     if currentBank == thisBank: # switch currentBank reference
89         currentBank = otherBank
90     else:
91         currentBank = thisBank
92
93 # boat has arrived
94 def boatArrived():
95
96     # check to see if the boat is at the shore
97     if AliceLiddell.DistanceTo( period ) < .01 or
98         AliceLiddell.DistanceTo( period2 ) < .01:
99         return 1
100     else:
101         return 0
102
103 # check to see if rules have been violated
104 def checkRules( currentBank ):
105
106     Animation1 = DoInOrder()
107     Animation2 = DoInOrder()
108
109     if "Chicken" in currentBank:
110
111         # chicken eats flower (i.e., seed)
112         if "Flower" in currentBank:
113             Animation1 = DoInOrder( camera.PointAt( Flower ),
114                                     Flower.destroy() )
115
116         # fox eats chicken
117         if "Fox" in currentBank:
118             Animation2 = DoInOrder( camera.PointAt( Chicken ),
119                                     Chicken.destroy() )
120
121         # player has lost
122         if ( "Flower" in currentBank ) or
123             ( "Fox" in currentBank ):
124             finishGame( Animation1, Animation2, GAMEOVER )
125
126         # player wins if nothing left on current bank
127         if len( currentBank ) == 0 and
128             not ( currentBank == targetBank ):
129             finishGame( Animation1, Animation2, CONGRATULATIONS )
130
131 # game over, AnimationX defaults to an empty sequence
132 def finishGame( Animation1, Animation2, final ):
133
134     controlPanel.destroy()
135     FollowTheBoat.stop()
136     final.Show()
137     DoInOrder( Animation1, Animation2,
138               DoInOrder( camera.Place( len( final._name ) + 2,
139                                     InFrontOf, final ), camera.PointAt( final ) ) )
140
141 # create control panel and buttons
142 controlPanel = AControlPanel( Caption = "Game Control Panel" )
143 animalListBox = \
144     controlPanel.MakeOptionButtonSet( List = thisBank[ : ],
145                                       Command = animalSelect )
146 buttonToBoat = \
147     controlPanel.MakeButton( Caption = "Get into the boat" )
148 buttonFromBoat = \
149     controlPanel.MakeButton( Caption = "Get out of the boat" )
150 buttonMoveBoat = \
151     controlPanel.MakeButton( Caption = "Go to the other shore" )
152
153 # associate buttons with callbacks
154 buttonToBoat.SetCommand( getIntoBoat )
155 buttonFromBoat.SetCommand( getOutOfBoat )
156 buttonMoveBoat.SetCommand( toOtherShore )
157

```

```

158 # initial selection defaults to first element (fox)
159 animalListbox.children[ 0 ].SetValue( 1 )
160 selected = "Fox"

```

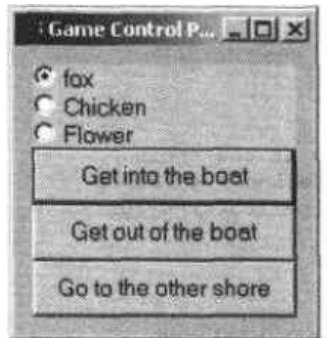


图 24.3 狐狸、鸡和种子游戏

最初的布景是用 Alice 环境及其预设对象创建的。这些对象包括 Alice Liddell、狐狸、鸡、花盆、河和船。Alice Liddell 的图像附加到船上。鸡、狐狸和种子最开始都在船边，位于同一个河岸。

布景中的其他所有对象只用于美化，不实际参与游戏。例如，第 15~22 行创建两条跳跃的鱼。程序使每条鱼移动和翻身（旋转），从而创建它的动画。第 16 行的 Move 方法取得三个参数，即运动方向、运动量（以米为单位）以及运动时间（以秒为单位）。在 LoopingFish 的情况下，对象要在 5 秒钟内向前运动 50 米。Turn 方法（第 17 行）也要取得三个参数，旋转方向、旋转量（旋转角度）以及旋转持续时间。LoopingFish 在 5 秒内向下旋转一圈。

在每次跳跃之间，我们都希望暂停一下，再显示鱼跳离水面的动画。所以在 Loop 调用内部（第 15~17 行），我们调用 AnimateWithPause 函数。Loop 函数连续调用作为它的一个参数传递的另一个函数。AnimateWithPause 函数（第 9~12 行）使一条鱼动起来，使每条鱼都在空中跳跃一下，并在指定的秒数之后重复这个动作。函数要取得两个参数：对象和时间（以秒为单位）。为每条鱼都调用这个函数。第 15~17 行和第 19~21 行将两个动画（运动和旋转）传给 animateWithPause。函数调用 DoTogether（第 11 行），后者取得两个动画，让一条鱼在运动的同时旋转。第 12 行强迫对象停止它的 forward 运动——调用 Move 函数时将运动量设为 0。DoInOrder 函数接收用逗号分隔的一个动画列表，并顺序地执行这些动画。DoInOrder 函数（第 11~12 行）取得对 DoTogether 和 Move 的调用，并顺序执行它们的结果。

现在，我们已经设置好布景并介绍了基本的动画原理，接着要创建一个控制面板来实现游戏操作。AControlPanel 函数（第 142 行）创建 controlPanel 窗口，以便在上面附加所有控制按钮。另外，将窗口的标题设为“Game Control Panel”（游戏控制面板）。第 143~145 行创建一系列单选钮，它们使用的是最初在岸边的一系列对象。每个单选钮都对应一个对象，即狐狸、鸡或者花盆（代表种子）。Command 选项将单选钮的选择与 animalSelect 函数关联到一起（第 33~36 行）。

我们在 controlPanel 上添加三个按钮，从而实现基本的游戏操作——将对象放到船上，让船离岸以及让一个对象下船。MakeButton 函数调用创建了三个按钮，即 buttonToBoat、buttonFromBoat 和 buttonMoveBoat。每个按钮上显示的标签都是指派给 Caption 的一个字符串。第 154~156 行调用 SetCommand 函数，将每个按钮都和它的事件处理程序关联到一起。最后，第 159~160 行将初始选择设为“fox”。

玩家在游戏中扮演 Alice，在两岸之间来回运动。第 5~6 行将照相机对准 Alice Liddell。船穿梭于两岸时，照相机将显示出游戏过程。Loop 函数确保照相机总是跟随 Alice Liddell 对象移动。

游戏状态用 Python 列表和变量来维护（第 24~30 行）。thisBank 列表引用游戏开始时的那个河岸。最初，有三个对象在列表中。otherBank 列表引用的是对岸，它最初是空的。currentBank 变量总是引用船所泊靠的河岸，而 targetBank 引用船的目标河岸。selected 变量（第 30 行）容纳着当前选择的对象。一旦选择和某个对象对应的单选钮（第 143~145 行），animalSelect 函数（第 33~36 行）就将 selected 设为选中的对象。注意 selected 是全局变量，其他函数也可访问它。

要将对象放到船上,需选择该对象的单选钮,再单击 Get into the boat 按钮(第146~147行),这样会调用 getIntoBoat 函数(第54~60行)。getIntoBoat 函数判断所选的对象能否放到船上。如对象在当前河岸(也就是船所泊靠的那个河岸),而且船上没有其他对象(第56~57行),函数就从当前河岸的列表中移除选中的对象,再把它添加到 theBoat。

第60行调用 ObjectInBoat 函数(第39~43行)将对象移动到船上。RespondToCollisionWith 函数(第41行)将 FishBoat 对触岸响应设为 Object.Stop。MoveTo 函数把对象移到船上,但它只能将对象设置到船的上方。要使对象落入船中,Move 函数需要使对象下落2米,以便与甲板相交。

如玩家希望将对象从船上转移到岸上,需要单击 Get out of the boat 按钮(第149~150行)。getOutOfBoat 函数(第63~68行)判断所选对象是否在船上,以及船是否到岸。如条件符合,就将对象从 Boat 中移除,再把它追加到当前河岸的列表中。

第68行调用 ObjectOutOfBoat 函数(定义于第46~51行),它将对象从船中移到岸上。对象根据其长度放到岸上(第49~51行);所以,每个对象都放到岸上一个不同的位置。第50~51行将对象移出船外,再让它下落至岸上。

用户选择 Go to the other shore 按钮后(第150~151行),toOtherShore 函数(第71~91行)会将船移到对岸。船行过程中,该函数会返回 None(第73~74行)。第76行使引用不同河岸的全局变量能在这个函数中访问。如船上有一个对象,对象会和船一起呈现动画效果(第78~81行)。船上的对象不是船的一部分,所以必须单独创建动画,并使对象位置和船的运动同步。DoInOrder 函数(第78~80行)使对象在3秒内前进16米,再使对象旋转半圈,并持续1秒钟。Turn 函数的第四个参数将 AsSeenby 设为 FishBoat。这样一来,当船靠岸时,对象会旋转成正对河岸;如果对象不和船一起旋转,对象会在旋转之后出现在船尾。第84~85行设置船的运动,使其遵循和对象一样的路径。

boatArrived 函数(第94~101行)判断船的状态。如果船正在运动,该函数返回0;否则返回1。这是借助位于两岸的两个小点(period)对象来实现的。通过检查 period 对象和 Alice Liddell 对象之间的距离,就能判断她是否抵达岸边。

第86行设置一个警报。经过指定时间后,警报会触发一个事件。Alice.SetAlarm 的第一个参数是要等待的时间,第二个参数是一个函数,超时就会调用该函数。在本例中,在船离岸的1秒钟之后,警报就会调用 checkRules 函数(第104~129行)。

checkRules 函数(第104~129行)判断是否违反一个规则,例如狐狸和鸡是否单独在同一个岸上。如违反规则,动画会相应变化,调用 finishGame 函数(第132~139行),并向该函数传递 GAMEOVER 参数。如果玩家输了,PointAt 方法会移动照相机,使其正对要被销毁的对象(即狐狸吃掉鸡,或者鸡吃掉花)。destroy 方法从布景上移除被“吃掉”的对象。如果没有违规,而且当前河岸为空,finishGame 会接收到 CONGRATULATIONS 参数,因为所有对象都被成功渡过河。

finishGame 函数(第132~139行)销毁 controlPanel,并停止船行动画。在游戏末尾,们显示 GAMEOVER 或 CONGRATULATIONS 字样。Place 函数(第138~139行)接受三个参数,相对于文本对象来放置照相机。这些参数是离开文本对象的距离,照相机的相对位置,以及具体的文本对象。本例将照相机放在文本对象前面,接着,PointAt 方法使照相机对准名为 final 的文本对象。

24.6 pygame 简介

24.7节、24.8节和24.9节要介绍 pygame,它是一系列由 Pete Shinnars 创作的 Python 模块,用于创建多媒体程序和游戏。pygame 模块利用了 Simple DirectMedia Layer (SDL)。SDL 是一个跨平台函数库,提供统一的 API 来访问多媒体硬件。pygame 模块允许程序员通过 Python 来访问这个库。要想了解 pygame 的详情(包括全面的用户文档),请访问 www.pygame.org。

24.7 Python CD Player

第一个例子演示如何创建简单的 CD Player 程序，这是使用 `pygame` 的 `cdrom` 模块来实现的，如图 24.4 所示。`cdrom` 模块包含 CD 类以及用于初始化 CD-ROM 子系统的方法。CD 类代表用户的 CD-ROM 驱动器。利用 CD 类的方法，用户可访问一张音乐 CD。图 24.4 的程序还使用 Tkinter 和 Pmw 来创建 CD Player 界面。Tkinter 和 Pmw 的详情参见第 10 章和第 11 章。

```

1 # Fig. 24.4: fig24_04.py
2 # Simple CD player using Tkinter and pygame.
3
4 import sys
5 import string
6 import pygame, pygame.cdrom
7 from Tkinter import *
8 from tkMessageBox import *
9 import Pmw
10
11 class CDPlayer( Frame ):
12     """A GUI CDPlayer class using Tkinter and pygame"""
13
14     def __init__( self ):
15         """Initialize pygame.cdrom and get CDROM if one exists"""
16
17         pygame.cdrom.init()
18
19         if pygame.cdrom.get_count() > 0:
20             self.CD = pygame.cdrom.CD( 0 )
21         else:
22             sys.exit( "There are no available CDROM drives." )
23
24         self.createGUI()
25         self.updateTime()
26
27     def destroy( self ):
28         """Stop CD, uninitialized pygame.cdrom and destroy GUI"""
29
30         if self.CD.get_init():
31             self.CD.stop()
32
33         pygame.cdrom.quit()
34         Frame.destroy( self )
35
36     def createGUI( self ):
37         """Create CDPlayer widgets"""
38
39         Frame.__init__( self )
40         self.pack( expand = YES, fill = BOTH )
41         self.master.title( "CD Player" )
42
43         # display current track playing
44         self.trackLabel = IntVar()
45         self.trackLabel.set( 1 )
46         self.trackDisplay = Label( self, font = "Courier 14",
47             textvariable = self.trackLabel, bg = "black",
48             fg = "green" )
49         self.trackDisplay.grid( sticky = W+E+N+S )
50
51         # display current time of track playing
52         self.timeLabel = StringVar()
53         self.timeLabel.set( "00:00/00:00" )
54         self.timeDisplay = Label( self, font = "Courier 14",
55             textvariable = self.timeLabel, bg = "black",
56             fg = "green" )
57         self.timeDisplay.grid( row = 0, column = 1, columnspan = 3,
58             sticky = W+E+N+S )
59

```

```

60     # play/pause CD
61     self.playLabel = StringVar()
62     self.playLabel.set( "Play" )
63     self.play = Button( self, textvariable = self.playLabel,
64         command = self.playCD, width = 10 )
65     self.play.grid( row = 1, column = 0, columnspan = 2,
66         sticky = W+E+N+S )
67
68     # stop CD
69     self.stop = Button( self, text = "Stop", width = 10,
70         command = self.stopCD )
71     self.stop.grid( row = 1, column = 2, columnspan = 2,
72         sticky = W+E+N+S )
73
74     # skip to previous track
75     self.previous = Button( self, text = "|<<", width = 5,
76         command = self.previousTrack )
77     self.previous.grid( row = 2, column = 0, sticky = W+E+N+S )
78
79     # skip to next track
80     self.next = Button( self, text = ">>|", width = 5,
81         command = self.nextTrack )
82     self.next.grid( row = 2, column = 1, sticky = W+E+N+S )
83
84     # eject CD
85     self.eject = Button( self, text = "Eject", width = 10,
86         command = self.ejectCD )
87     self.eject.grid( row = 2, column = 2, columnspan = 2,
88         sticky = W+E+N+S )
89
90     def playCD( self ):
91         """Play/Pause CD if disc is loaded"""
92
93         # if disc has been ejected, reinitialize drive
94         if not self.CD.get_init():
95             self.CD.init()
96             self.currentTrack = 1
97
98         # if no disc in drive, uninitialize and return
99         if self.CD.get_empty():
100             self.CD.quit()
101             return
102
103         # if disc is loaded, obtain disc information
104         else:
105             self.totalTracks = self.CD.get_numtracks()
106
107         # if CD is not playing, play CD
108         if not self.CD.get_busy() and not self.CD.get_paused():
109             self.CD.play( self.currentTrack - 1 )
110             self.playLabel.set( "| |" )
111
112         # if CD is playing, pause disc
113         elif not self.CD.get_paused():
114             self.CD.pause()
115             self.playLabel.set( "Play" )
116
117         # if CD is paused, resume play
118         else:
119             self.CD.resume()
120             self.playLabel.set( "| |" )
121
122     def stopCD( self ):
123         """Stop CD if disc is loaded"""
124
125         if self.CD.get_init():
126             self.CD.stop()
127             self.playLabel.set( "Play" )
128
129     def playTrack( self, track ):
130         """Play track if disc is loaded"""

```

```

131
132     if self.CD.get_init():
133         self.currentTrack = track
134         self.trackLabel.set( self.currentTrack )
135
136         # start beginning of track
137         if self.CD.get_busy():
138             self.CD.play( self.currentTrack - 1 )
139         elif self.CD.get_paused():
140             self.CD.play( self.currentTrack - 1 )
141             self.playCD() # re-pause CD
142
143     def nextTrack( self ):
144         """Play next track on CD if disc is loaded"""
145
146         if self.CD.get_init() and \
147            self.currentTrack < self.totalTracks:
148             self.playTrack( self.currentTrack - 1 )
149
150     def previousTrack( self ):
151         """Play previous track on CD if disc is loaded"""
152
153         if self.CD.get_init() and self.currentTrack > 1:
154             self.playTrack( self.currentTrack - 1 )
155
156     def ejectCD( self ):
157         """Eject CD from drive"""
158
159         response = askyesno( "Eject pushed", "Eject CD?" )
160
161         if response:
162             self.CD.init() # CD must be initialized to eject
163             self.CD.eject()
164             self.CD.quit()
165             self.trackLabel.set( 1 )
166             self.timeLabel.set( "00:00/00:00" )
167             self.playLabel.set( "Play" )
168
169     def updateTime( self ):
170         """Update time display if disc is loaded"""
171
172         if self.CD.get_init():
173             seconds = int( self.CD.get_current()[ 1 ] )
174             endSeconds = int( self.CD.get_track_length(
175                 self.currentTrack - 1 ) )
176
177             # if reached end of current track, play next track
178             if seconds >= ( endSeconds - 1 ):
179                 self.nextTrack()
180             else:
181                 minutes = seconds / 60
182                 endMinutes = endSeconds / 60
183                 seconds = seconds - ( minutes * 60 )
184                 endSeconds = endSeconds - ( endMinutes * 60 )
185
186                 # display time in format mm:ss/mm:ss
187                 trackTime = string.zfill( str( minutes ), 2 ) + \
188                     ":" + string.zfill( str( seconds ), 2 )
189                 endTime = string.zfill( str( endMinutes ), 2 ) + \
190                     ":" + string.zfill( str( endSeconds ), 2 )
191
192                 if self.CD.get_paused():
193
194                     # alternate pause symbol and time in display
195                     if not self.timeLabel.get() == " | ":
196                         self.timeLabel.set( " | " )
197                     else:
198                         self.timeLabel.set( trackTime + "/" + endTime )
199
200             else:
201                 self.timeLabel.set( trackTime + "/" + endTime )

```

```

202
203     # call updateTime method again after 1000ms ( 1 second )
204     self.after( 1000, self.updateTime )
205
206 def main():
207     CDPlayer().mainloop()
208
209 if __name__ == "__main__":
210     main()

```



图 24.4 Python CD Player

第 207 行创建一个 CDPlayer 对象，并调用其 mainloop 方法来启动应用程序。CDPlayer 构造函数（第 14~25 行）初始化 cdrom 模块（第 17 行）。初始化 cdrom 模块后，程序就可使用各种方法来控制和查询任何 CD-ROM 驱动器。if/else 结构（第 19~22 行）调用 cdrom 的 get_count 方法来判断计算机上可用的 CD-ROM 驱动器。get_count 方法返回的是计算机上可用的 CD-ROM 驱动器数量。如果至少存在一部驱动器，第 20 行就创建名为 CD 的一个 CD 对象。传给 CD 构造函数的值是 CD-ROM 驱动器的标识（ID）号。如果同时安装了多部 CD-ROM 驱动器，程序将选用主驱动器。构造函数接收一个 0 作为参数，因为主 CD-ROM 的驱动器标识号肯定是 0。如果不存在 CD-ROM 驱动器，程序退出（第 22 行）。

程序识别出存在一部 CD-ROM 驱动器后，就构造一个便于用户交互的 GUI。第 24 行调用 createGUI 方法来创建 CD Player 图形界面，其中含有各种不同的 GUI 组件。createGUI 方法（第 36~88 行）将各个组件添加到显示区域（稍后会讲解每个组件的动作）。用于显示曲目编号的标签（trackDisplay）以及用于显示当前曲目时间的标签（timeDisplay）都使用了 textvariable（文本变量），分别是 trackLabel 和 timeLabel，它们可对 CD Player 的显示进行更新。注意 play 按钮也有一个 textvariable，即 playLabel，它负责在暂停或播放 CD 时改变按钮上的显示。

创建好 GUI 后，构造函数要调用 updateTime 方法（稍后讨论）。接着，程序进入 mainloop，可在其中进行播放、停止、暂停、快进和快退操作。

其他方法则提供了一个基本 CD Player 的全部功能。Play 按钮的回调方法是 playCD（第 90~120 行），它可播放或暂停 CD。第 94 行调用 CD 的 get_init 方法，判断 CD-ROM 是否被初始化。如果尚未初始化，playCD 会初始化它，并将 currentTrack 设为 1。currentTrack 变量存储的是当前曲目的编号。第 99 行调用 CD 的 get_empty 方法，判断驱动器是不是空的。如果是，第 100 行就使用 CD 的 quit 方法撤消对象初始化并返回。否则，第 105 行调用 CD 的 get_numtracks 方法来获得光盘上的总曲目数，并将该值存储到对象的 totalTracks 属性中。

playCD 方法检测三种情况——CD 当前没有播放，CD 不是暂停，以及 CD 是在暂停。第 108 行判断 CD 是否没有播放而且不是暂停（分别使用 get_busy 和 get_paused 方法）。如果两个条件都为 true，第 109 行调用 play 方法。在方法调用中指定了要播放哪个曲目（track）。由于 CD 对象的曲目编号开始于 0，而且 currentTrack 初始化成 1，所以传给 play 方法的值要比 currentTrack 小 1（第 109 行）。第 110 行设置 Play 按钮的文本，使其包含用于暂停标志 |。这就使用户能够正确地控制 CD-ROM 应用程序。

第 113 行判断 CD 是否暂停，这是通过调用 get_paused 方法来实现的。如果 CD 正在播放而且没有暂停（用户没有按下暂停按钮），CD 的 pause 方法（第 114 行）就暂停 CD 播放。第 114 行设置 Play 按钮的文本，使其包含 Play 字样。否则就表明 CD 当前是暂停的，所以调用 CD 的 resume 方法（第 119 行）来继续播放。在第 110 行中，程序将按钮标签设置与 Paused 按钮对应的符号。

与 Stop 按钮对应的回调方法是 stopCD（第 122~127 行）。按下这个按钮后，第 125 行判断 CD 对象是否初始化。如果是，就调用 CD 的 stop 方法来停止 CD 播放，Play 按钮再次显示“Play”字样。如果

CD 当前不在播放, 那么调用 `stop` 不会发生任何事情。但是, 第 125 行仍要判断 CD 对象是否初始化, 否则 `stop` 会产生一个错误。

这个程序顺序播放曲目, 但用户可按|<<或>>|按钮切换到上一曲或下一曲。与>>|按钮对应的回调方法是 `nextTrack`, 它切换到 CD 上的下一曲 (第 143~148 行)。如果 CD 已经初始化, 而且当前曲目不是最后一个曲目, 就调用 `playTrack` 方法, 并向其传递下一曲的编号 (`currentTrack + 1`)。类似地, 与|<<按钮对应的回调方法是 `previousTrack`, 它切换到上一曲 (第 150~154 行)。如果 CD 已经初始化, 而且当前曲目不是第一个曲目, 就调用 `playTrack` 方法, 并向其传递上一曲的编号 (`currentTrack - 1`)。

`playTrack` 方法 (第 129~141 行) 负责播放一个曲目。如果 CD 已经初始化, 第 133 行就将 `currentTrack` 设为指定的曲目编号。第 134 行将 `trackLabel` 设为新的曲目编号。如果 CD 正在播放另一曲, 第 138 行就改为播放指定的曲目。但假如 CD 当前处于暂停状态, 第 140~141 行就切换到指定的曲目, 但依旧暂停。

为 Eject (弹出) 按钮绑定的回调方法是 `ejectCD` (第 156~167 行)。一旦单击该按钮, 第 159 行就显示一个 `tkMessageBox` 窗口, 用一条消息询问用户是否真的想弹出。这是防止偶然弹出 CD 的一个安全措施。如选择弹出 CD, 就初始化 CD, 因为假如用户从未播放过 CD, 它就从未被初始化过。而试图弹出一个未初始化的 CD 对象会导致错误。CD 对象初始化后, 就用 CD 的 `eject` 方法来弹出光盘, 同时撤消对 CD 对象的初始化 (第 162~164 行)。第 165~167 行将 CD Player 的界面设置成最开始的样子。

CD Player 使用 `updateTime` 方法 (第 169~204 行) 来更新显示, 该方法最初是在第 25 行调用的。第 172 行判断 CD 是否初始化。如果不是, 跳到第 204 行执行。

通常, 音乐 CD 会列出每个曲目的持续时间。该程序会将一首歌曲已经播放的时间显示出来。如果 CD 已经初始化, CD 的 `get_current` 方法就会返回曲目已经播放的秒数, 我们将该值指派给变量 `seconds` (第 173 行)。`get_current` 方法返回的是一个包含两个元素的元组, 包括当前曲目编号以及已经播放的秒数。第 174~175 行从 CD 的 `get_track_length` 方法获得曲目长度, 并将当前曲目指定为 (`currentTrack - 1`)。曲目长度值要指派给变量 `endSeconds`。第 178~179 行保证各个曲目连续地播放, 直到播放完所有曲目。第 181~184 行使用 `seconds` 和 `endSeconds` 来判断当时时间和结束时间 (使用分钟数和秒数)。

第 187~188 行为当前曲目时间 (`trackTime`) 创建一个字符串。字符串的形式为 `mm:ss`, 其中的 `mm` 代表分钟数, `ss` 代表秒数。注意 `string` 函数 `zfill` 可在字符串中填充零, 以便凑足正确的长度。这就保证分钟和秒都显示成两位。

第 192 行判断 CD 是否已暂停。如果不是, 就更新 `timeDisplay` 以显示当前时间 (第 201 行)。否则, 就将 `timeDisplay` 更新为当前时间或者代表暂停的符号 (第 195~198 行)。暂停时, 会因为曲目时间和暂停符号的切换而导致显示的闪烁。

`updateTime` 方法调用组件方法 `after`。`after` 方法注册一个回调方法, 以便在经过指定的毫秒数之后执行。第 204 行保证每隔约 1000 毫秒 (1 秒) 就调用一次 `updateTime` 方法。

结束 CD Player 的执行时, 程序会销毁窗口, 并调用 CDPlayer 的 `destroy` 方法来终止播放程序 (第 27~34 行)。第 30 行判断 CD 是否初始化。如果初始化, 就调用 CD 的 `stop` 方法来停止播放。如果不调用 `stop` 方法, 在用户销毁了窗口后, CD 会继续播放。第 33~34 行撤消对 `pygame` 的 `cdrom` 模块的初始化, 并调用 `Frame` 的 `destroy` 方法以销毁帧。

良好编程习惯 24.1 对于 Tkinter 程序, `destroy` 方法相当于析构函数。

24.8 Python Movie Player

本节创建一个简单电影播放程序来演示 `pygame` 的 `movie` 模块 (图 24.5)。`movie` 模块的一个方法可创建 `Movie` 对象, 它代表一个打开的 MPEG 文件。`Movie` 类提供了用于播放、停止、暂停和倒带的方法。

```
1 # Fig. 24.5: fig24_05.py
2 # Playing an MPEG movie.
```



```

3
4 import os
5 import sys
6 import pygame, pygame.movie
7 import pygame.mouse, pygame.image
8 from pygame.locals import *
9
10 def createGUI( file ):
11
12     # load movie
13     movie = pygame.movie.Movie( file )
14     width, height = movie.get_size()
15
16     # initialize display window
17     screen = pygame.display.set_mode( ( width, height + 100 ) )
18     pygame.display.set_caption( "Movie Player" )
19     pygame.mouse.set_visible( 1 )
20
21     # play button
22     playImageFile = os.path.join( "data", "play.png" )
23     playImage = pygame.image.load( playImageFile ).convert()
24     playImageSize = playImage.get_rect()
25     playImageSize.center = width / 2, height + 50
26
27     # copy play button to screen
28     screen.blit( playImage, playImageSize )
29     pygame.display.flip()
30
31     # set output surface for movie's video
32     movie.set_display( screen )
33
34     return movie, playImageSize
35
36 def main():
37
38     # check command line arguments
39     if len( sys.argv ) != 2:
40         sys.exit( "Incorrect number of arguments." )
41     else:
42         file = sys.argv[ 1 ]
43
44     # initialize pygame
45     pygame.init()
46
47     # initialize GUI
48     movie, playImageSize = createGUI( file )
49
50     # wait until player wants to close program
51     while 1:
52         event = pygame.event.wait()
53
54         # close window
55         if event.type == QUIT or \
56            ( event.type == KEYDOWN and event.key == K_ESCAPE ):
57             break
58
59         # click play button and play movie
60         pressed = pygame.mouse.get_pressed()[ 0 ]
61         position = pygame.mouse.get_pos()
62
63         # button pressed
64         if pressed:
65
66             if playImageSize.collidepoint( position ):
67                 movie.play()
68
69 if __name__ == "__main__":
70     main()

```



图 24.5 Python Movie Player

程序运行时会执行 `main` 函数（第 36~67 行）。第 39~42 行判断用户是否提供了正确数量的命令行参数。为了运行这个例子，用户需要提供一个命令行参数来指定电影文件的位置，例如：

```
python fig24_05.py "c:\bailey.mpg"
```

电影文件 `bailey.mpg` 可从 www.deitel.com 下载。如果没有提供文件名，会显示一条错误消息。

第 45 行初始化 `pygame`。`pygame.init` 方法初始化每个 `pygame` 模块，其中包括本例要使用的，即 `pygame.movie`、`pygame.mouse` 和 `pygame.image`。这样调用 `init` 是一种快捷手段，程序就不必单独调用每个模块的 `init` 方法。第 49 行调用 `createGUI` 方法（在第 10~34 行定义），以便为应用程序初始化 GUI，并获得电影对象和播放按钮。

`createGUI` 方法（第 10~34 行）载入 MPEG 电影，并生成显示窗口。第 13 行调用 `pygame.movie.Movie` 方法来创建一个 `Movie` 对象。`pygame.movie.Movie` 方法要取得一个参数，即电影文件名。`Movie` 对象的 `get_size` 方法（第 14 行）以元组形式返回电影大小，元组中包含宽度和高度。

程序（第 17~19 行）初始化显示窗口，使其能容下电影和一个播放按钮。第 17 行使用 `pygame.display` 的 `set_mode` 方法来设置当前窗口大小。传给 `set_mode` 的参数是一个含有两个元素的元组，它将窗口大小的宽度设为 `width`，高度设为 `(height + 100)`，单位都是像素。`set_mode` 的返回值是 `pygame` 的一个 `Surface` 对象，这是一个空白画布，程序将在其中显示电影。`Surface` 对象被指派给变量 `screen`。第 18 行调用 `pygame.display` 的 `set_caption` 方法，将窗口标题设为 "Movie Player"。

用户单击播放按钮开始播放电影。`createGUI` 方法允许用户在显示中看到鼠标指针，并将播放按钮放到 GUI 上。第 19 行调用 `pygame.mouse` 的 `set_visible` 方法，并传递参数 1，使鼠标指针出现在窗口上方。播放按钮的图像在 "data" 子目录中。`os.path.join` 方法（第 22 行）要取得两个参数，即子目录和图像文件名 "play.png"。该方法采取特定于系统的方式合并子目录和文件名，从而定位文件。例如在 Windows 系统的文件目录中，路径是 "data\play.png"。返回的图像指派给 `playImageFile`。

第 23 行 `pygame.image` 的 `load` 方法载入播放按钮图像，而 `convert` 方法创建 `Surface` 对象的一个拷贝，并将按钮图像的像素格式转换成显示区域的像素格式。格式相同的像素是用数量相同的二进制位来表示的。将表面和按钮设为相同像素格式，使程序能够更快地叠置（`blit`）不同的表面。`load` 方法将转换好的按钮放到 `pygame Surface` 上。

需要准确指定图像放在显示区域的什么位置，为此可利用 `Rect` 类。第 24 行调用 `Surface` 对象 `playImage` 的 `get_rect` 方法，从而获得播放按钮图像大小。`get_rect` 返回的对象就是一个 `Rect` 对象（即一个 `pygame` 矩形），它被指派给 `playImageSize`。`Rect` 对象代表覆盖播放按钮图像整个表面的一个矩形。第 25 行通过

修改播放按钮图像的 center 属性来设置播放按钮的位置。

最后将播放按钮图像添加到显示区域。第 28 行在屏幕上的 playImageSize 位置描绘播放按钮图像。位置(0,0)代表屏幕左上角。第 29 行调用 pygame.display 模块的 flip 方法来更新显示。flip 方法更新整个显示表面。下一节将讨论为什么说这并非肯定必要(或有效)。

接着,第 32 行设置显示区域,使电影在 Surface 的 screen 对象上呈现。Movie 类的 set_display 方法取得两个参数,即为电影输出的 Surface 对象,以及电影左上角位置。第二个参数是可选的。如果未指定,就使用默认值(0,0)。第 34 行返回 movie 对象以及 playImageSize 矩形,后者代表播放按钮。

控制权回到 main 函数后, Surface 的 screen 对象上最初显示的是位于空白屏幕下的一个圆形播放按钮。while 循环等待用户开始播放电影或关闭应用程序(第 51~67 行)。第 52 行调用 pygame.event 的 wait 方法,它等待并返回队列中等待的下一个 Event 对象。如事件是请求退出程序(QUIT),或者是一个 KEYDOWN 事件(key 属性等于 K_ESCAPE),那么程序退出 while 循环(第 57 行),应用程序终止。

为播放电影,用户可单击鼠标右键。第 60 行判断用户是否单击鼠标右键。pygame.mouse.get_pressed 方法返回 3 个值的一个序列,它们指出单击的是鼠标左键、中键还是右键。鼠标右键的状态是序列中的第一个项目。如果按下鼠标右键,该值为 true。

程序需要检测用户是否按下播放按钮。Rect 对象 playImageSize 提供这一功能。首先,第 61 行调用 pygame.mouse.get_pos 方法来获得鼠标指针位置。如按下鼠标右键(第 66 行),Rect 类的 collidepoint 方法(第 67 行)判断单击位置是否在按钮内部或边框上。该方法取得一个参数,也就是要检测的点的位置。如这个点在矩形内部,或在它的边框上,方法返回 true。第 67 行调用 Movie 类的 play 方法来播放电影。

24.9 用 pygame 开发太空船游戏

尽管也可用 pygame 开发其他类型的程序,但它最常见的应用还是游戏。图 24.6 使用各种 pygame 模块创建一个简单的“Space Cruiser”(太空船)游戏,玩家有 60 秒时间将太空船驶过一个小行星带。60 秒后,能量耗尽,游戏结束。屏幕左上角的时钟显示剩余时间。飞船撞到一个小行星,剩余时间减 5 秒。但也有机会拾到能量块。每个能量块为计时器添加 5 秒的时间。玩家使用键盘上的箭头键控制飞船。

```

1 # Fig. 24.6: fig24_06.py
2 # Space Cruiser game using pygame.
3
4 import os
5 import sys
6 import random
7 import pygame, pygame.image, pygame.font, pygame.mixer
8 from pygame.locals import *
9
10 class Sprite:
11     """An object to place on the screen"""
12
13     def __init__( self, image ):
14         """Initialize object image and calculate rectangle"""
15
16         self.image = image
17         self.rectangle = image.get_rect()
18
19     def place( self, screen ):
20         """Place object on the screen"""
21
22         return screen.blit( self.image, self.rectangle )
23
24     def remove( self, screen, background ):
25         """Place background over image to remove it"""
26
27         return screen.blit( background, self.rectangle,
28                             self.rectangle )
29
30 class Player( Sprite ):

```

```
31 """Player Sprite with 4 different states"""
32
33 def __init__( self, images, crashImage,
34             centerX = 0, centerY = 0 ):
35     """Store images and set initial Player state"""
36
37     self.movingImages = images
38     self.crashImage = crashImage
39     self.centerX = centerX
40     self.centerY = centerY
41     self.playerPosition = 1 # start player facing down
42     self.speed = 0
43     self.loadImage()
44
45 def loadImage( self ):
46     """Load Player image and calculate rectangle"""
47
48     if self.playerPosition == -1: # player has crashed
49         image = self.crashImage
50     else:
51         image = self.movingImages[ self.playerPosition ]
52
53     Sprite.__init__( self, image )
54     self.rectangle.centerX = self.centerX
55     self.rectangle.centerY = self.centerY
56
57 def moveLeft( self ):
58     """Change Player image to face one position to left"""
59
60     if self.playerPosition == -1: # player crashed
61         self.speed = 1
62         self.playerPosition = 0 # move left of obstacle
63     elif self.playerPosition > 0:
64         self.playerPosition -= 1
65
66     self.loadImage()
67
68 def moveRight( self ):
69     """Change Player image to face one position to right"""
70
71     if self.playerPosition == -1: # player crashed
72         self.speed = 1
73         self.playerPosition = 2 # move right of obstacle
74     elif self.playerPosition < ( len( self.movingImages ) - 1 ):
75         self.playerPosition += 1
76
77     self.loadImage()
78
79 def decreaseSpeed( self ):
80
81     if self.speed > 0:
82         self.speed -= 1
83
84 def increaseSpeed( self ):
85
86     if self.speed < 10:
87         self.speed += 1
88
89     # player crashed, start player facing down
90     if self.playerPosition == -1:
91         self.playerPosition = 1
92         self.loadImage()
93
94 def collision( self ):
95     """Change Player image to crashed player"""
96
97     self.speed = 0
98     self.playerPosition = -1
99     self.loadImage()
100
101 def collisionBox( self ):
```

```

102     """Return smaller bounding box for collision tests"""
103
104     return self.rectangle.inflate( -20, -20 )
105
106 def isMoving( self ):
107     """Player is not moving if speed is 0"""
108
109     if self.speed == 0:
110         return 0
111     else:
112         return 1
113
114 def distanceMoved( self ):
115     """Player moves twice as fast when facing straight down"""
116
117     xIncrement, yIncrement = 0, 0
118
119     if self.isMoving():
120
121         if self.playerPosition == 1:
122             xIncrement = 0
123             yIncrement = 2 * self.speed
124         else:
125             xIncrement = ( self.playerPosition - 1 ) * self.speed
126             yIncrement = self.speed
127
128     return xIncrement, yIncrement
129
130 class Obstacle( Sprite ):
131     """Moveable Obstacle Sprite"""
132
133     def __init__( self, image, centerX = 0, centerY = 0 ):
134         """Load Obstacle image and initialize rectangle"""
135
136         Sprite.__init__( self, image )
137
138         # move Obstacle to specified location
139         self.positiveRectangle = self.rectangle
140         self.positiveRectangle.centerX = centerX
141         self.positiveRectangle.centery = centerY
142
143         # display Obstacle in moved position to buffer visible area
144         self.rectangle = self.positiveRectangle.move( -60, -60 )
145
146     def move( self, xIncrement, yIncrement ):
147         """Move Obstacle location up by specified increments"""
148
149         self.positiveRectangle.centerX -= xIncrement
150         self.positiveRectangle.centery -= yIncrement
151
152         # change position for next pass
153         if self.positiveRectangle.centery < 25:
154             self.positiveRectangle[ 0 ] += \
155                 random.randrange( -640, 640 )
156
157         # keep rectangle values from overflowing
158         self.positiveRectangle[ 0 ] %= 760
159         self.positiveRectangle[ 1 ] %= 600
160
161         # display obstacle in moved position to buffer visible area
162         self.rectangle = self.positiveRectangle.move( -60, -60 )
163
164     def collisionBox( self ):
165         """Return smaller bounding box for collision tests"""
166
167         return self.rectangle.inflate( -20, -20 )
168
169 class Objective( Sprite ):
170     """Moveable Objective Sprite"""
171
172     def __init__( self, image, centerX = 0, centerY = 0 ):

```

```

173     """Load Objective image and initialize rectangle"""
174
175     Sprite.__init__(self, image)
176
177     # move Objective to specified location
178     self.rectangle.centerx = centerX
179     self.rectangle.centery = centerY
180
181     def move(self, xIncrement, yIncrement):
182         """Move Objective location up by specified increments"""
183
184         self.rectangle.centerx -= xIncrement
185         self.rectangle.centery -= yIncrement
186
187     # place a message on screen
188     def displayMessage(message, screen, background):
189         font = pygame.font.Font(None, 48)
190         text = font.render(message, 1, (250, 250, 250))
191         textPosition = text.get_rect()
192         textPosition.centerx = background.get_rect().centerx
193         textPosition.centery = background.get_rect().centery
194         return screen.blit(text, textPosition)
195
196     # remove outdated time display and place updated time on screen
197     def updateClock(time, screen, background, oldPosition):
198         remove = screen.blit(background, oldPosition, oldPosition)
199         font = pygame.font.Font(None, 48)
200         text = font.render(str(time), 1, (250, 250, 250),
201                             (0, 0, 0))
202         textPosition = text.get_rect()
203         post = screen.blit(text, textPosition)
204         return remove, post
205
206     def main():
207
208         # constants
209         WAIT_TIME = 20          # time to wait between frames
210         COURSE_DEPTH = 50 * 480  # 50 screens long
211         NUMBER_ASTEROIDS = 20    # controls number of asteroids
212
213         # variables
214         distanceTraveled = 0     # vertical distance
215         nextTime = 0            # time to generate next frame
216         courseOver = 0          # the course has not been completed
217         allAsteroids = []       # randomly generated obstacles
218         dirtyRectangles = []    # screen positions that have changed
219         energyPack = None       # current energy pack on screen
220         timeLeft = 60           # time left to finish course
221         newClock = (0, 0, 0, 0) # location of clock
222
223         # find path to sounds
224         collisionFile = os.path.join("data", "collision.wav")
225         chimeFile = os.path.join("data", "energy.wav")
226         startFile = os.path.join("data", "toneup.wav")
227         applauseFile = os.path.join("data", "applause.wav")
228         gameOverFile = os.path.join("data", "tonedown.wav")
229
230         # find path to images
231         shipFiles = []
232         shipFiles.append(os.path.join("data", "shipLeft.gif"))
233         shipFiles.append(os.path.join("data", "shipDown.gif"))
234         shipFiles.append(os.path.join("data", "shipRight.gif"))
235         shipCrashFile = os.path.join("data", "shipCrashed.gif")
236         asteroidFile = os.path.join("data", "Asteroid.gif")
237         energyPackFile = os.path.join("data", "Energy.gif")
238
239         # obtain user preference
240         fullScreen = int(raw_input(
241             "Fullscreen? (0 = no, 1 = yes): "))
242
243         # initialize pygame

```

```

244     pygame.init()
245
246     if fullScreen:
247         screen = pygame.display.set_mode( ( 640, 480 ), FULLSCREEN )
248     else:
249         screen = pygame.display.set_mode( ( 640, 480 ) )
250
251     pygame.display.set_caption( "Space Cruiser!" )
252     pygame.mouse.set_visible( 0 ) # make mouse invisible
253
254     # create background and fill with black
255     background = pygame.Surface( screen.get_size() ).convert()
256
257     # blit background onto screen and update entire display
258     screen.blit( background, ( 0, 0 ) )
259     pygame.display.update()
260
261     collisionSound = pygame.mixer.Sound( collisionFile )
262     chimeSound = pygame.mixer.Sound( chimeFile )
263     startSound = pygame.mixer.Sound( startFile )
264     applauseSound = pygame.mixer.Sound( applauseFile )
265     gameOverSound = pygame.mixer.Sound( gameOverFile )
266
267     # load images, convert pixel format and make white transparent
268     loadedImages = []
269
270     for file in shipFiles:
271         surface = pygame.image.load( file ).convert()
272         surface.set_colorkey( surface.get_at( ( 0, 0 ) ) )
273         loadedImages.append( surface )
274
275     # Load crash image
276     shipCrashImage = pygame.image.load( shipCrashFile ).convert()
277     shipCrashImage.set_colorkey( shipCrashImage.get_at( ( 0, 0 ) ) )
278
279     # initialize theShip
280     centerX = screen.get_width() / 2
281     theShip = Player( loadedImages, shipCrashImage, centerX, 25 )
282
283     # load asteroid image
284     asteroidImage = pygame.image.load( asteroidFile ).convert()
285     asteroidImage.set_colorkey( asteroidImage.get_at( ( 0, 0 ) ) )
286
287     # place asteroid in randomly generated spot
288     for i in range( NUMBER_ASTEROIDS ):
289         allAsteroids.append( Obstacle( asteroidImage,
290             random.randrange( 0, 760 ), random.randrange( 0, 600 ) ) )
291
292     # load energyPack image
293     energyPackImage = pygame.image.load( energyPackFile ).convert()
294     energyPackImage.set_colorkey( surface.get_at( ( 0, 0 ) ) )
295
296     startSound.play()
297     pygame.time.set_timer( USEREVENT, 1000 )
298
299     while not courseOver:
300
301         # wait if moving too fast for selected frame rate
302         currentTime = pygame.time.get_ticks()
303
304         if currentTime < nextTime:
305             pygame.time.delay( nextTime - currentTime )
306
307         nextTime = currentTime + WAIT_TIME
308
309         # remove objects from screen
310         dirtyRectangles.append( theShip.remove( screen,
311             background ) )
312
313         for asteroid in allAsteroids:
314             dirtyRectangles.append( asteroid.remove( screen,

```

```

315         background ) )
316
317     if energyPack is not None:
318         dirtyRectangles.append( energyPack.remove( screen,
319             background ) )
320
321     # get next event from event queue
322     event = pygame.event.poll()
323
324     # if player quits program or presses escape key
325     if event.type == QUIT or \
326         ( event.type == KEYDOWN and event.key == K_ESCAPE ):
327         sys.exit()
328
329     # if up arrow key was pressed, slow ship
330     elif event.type == KEYDOWN and event.key == K_UP:
331         theShip.decreaseSpeed()
332
333     # if down arrow key was pressed, speed up ship
334     elif event.type == KEYDOWN and event.key == K_DOWN:
335         theShip.increaseSpeed()
336
337     # if right arrow key was pressed, move ship right
338     elif event.type == KEYDOWN and event.key == K_RIGHT:
339         theShip.moveRight()
340
341     # if left arrow key was pressed, move ship left
342     elif event.type == KEYDOWN and event.key == K_LEFT:
343         theShip.moveLeft()
344
345     # one second has passed
346     elif event.type == USEREVENT:
347         timeLeft -= 1
348
349     # 1 in 100 odds of creating new energyPack
350     if energyPack is None and not random.randrange( 100 ):
351         energyPack = Objective( energyPackImage,
352             random.randrange( 0, 640 ), 480 )
353
354     # update obstacle and energyPack positions if ship moving
355     if theShip.isMoving():
356         xIncrement, yIncrement = theShip.distanceMoved()
357
358         for asteroid in allAsteroids:
359             asteroid.move( xIncrement, yIncrement )
360
361         if energyPack is not None:
362             energyPack.move( xIncrement, yIncrement )
363
364             if energyPack.rectangle.bottom < 0:
365                 energyPack = None
366
367         distanceTraveled += yIncrement
368
369     # check for collisions with smaller bounding boxes
370     # for better playability
371     asteroidBoxes = []
372
373     for asteroid in allAsteroids:
374         asteroidBoxes.append( asteroid.collisionBox() )
375
376     # retrieve list of obstacles colliding with player
377     collision = theShip.collisionBox().collidelist(
378         asteroidBoxes )
379
380     # move asteroid one screen down
381     if collision != -1:
382         collisionSound.play()
383         allAsteroids[ collision ].move( 0, -540 )
384         theShip.collision()
385         timeLeft -= 5

```



```

386
387     # determine whether player has gotten energyPack
388     if energyPack is not None:
389
390         if theShip.collisionBox().collidirect(
391             energyPack.rectangle ):
392             chimeSound.play()
393             energyPack = None
394             timeLeft += 5
395
396     # place objects on screen
397     dirtyRectangles.append( theShip.place( screen ) )
398
399     for asteroid in allAsteroids:
400         dirtyRectangles.append( asteroid.place( screen ) )
401
402     if energyPack is not None:
403         dirtyRectangles.append( energyPack.place( screen ) )
404
405     # update time
406     oldClock, newClock = updateClock( timeLeft, screen,
407         background, newClock )
408     dirtyRectangles.append( oldClock )
409     dirtyRectangles.append( newClock )
410
411     # update changed areas of display
412     pygame.display.update( dirtyRectangles )
413     dirtyRectangles = []
414
415     # check for course end
416     if distanceTraveled > COURSE_DEPTH:
417         courseOver = 1
418
419     # check for game over
420     elif timeLeft <= 0:
421         break
422
423     if courseOver:
424         applauseSound.play()
425         message = "Asteroid Field Crossed!"
426     else:
427         gameOverSound.play()
428         message = "Game Over!"
429
430     pygame.display.update( displayMessage( message, screen,
431         background ) )
432
433     # wait until player wants to close program
434     while 1:
435         event = pygame.event.poll()
436
437         if event.type == QUIT or \
438             ( event.type == KEYDOWN and event.key == K_ESCAPE ):
439             break
440
441 if __name__ == "__main__":
442     main()

```

图 24.6 用 pygame 实现的太空船游戏

程序运行时，会执行 main 函数（第 206~439 行）。第 209~221 行创建程序使用的常量和变量。后面会对其详加解释。第 224~237 行在"data"子目录中定位声音和图像文件。程序提示玩家选择全屏或窗口模式。全屏模式将游戏窗口设置成整个屏幕大小；而窗口模式使用一个较的游戏显示区域。玩家的响应被指派给 fullScreen 变量，0 代表窗口模式，1 代表全屏模式。

第 244 行初始化 pygame。这个 init 调用避免了单独调用每个模块的 init 方法的必要。第 246~249 行使用 pygame.display 的 set_mode 方法设置当前显示模式。传给 set_mode 的第一个参数是包含两个元素

的一个元组，它指定 640 像素宽、480 像素高的一个显示区域。如果玩家选择全屏模式，程序就为 `FULLSCREEN` 常量传递 `set_mode`。如果玩家不选择全屏模式，程序就不向方法传递参数。`set_mode` 方法返回一个 `pygame Surface` 对象，它是一个空白画布，便于程序在上面绘制游戏画面。这个 `Surface` 对象被指派给 `screen` 变量。第 251 行将窗口标题设为 "Space Cruiser!"，这是通过调用 `pygame.display` 的 `set_caption` 方法来实现的。第 252 行调用 `pygame.mouse` 的 `set_visible` 方法，并向其传递参数 0，表明鼠标指针不出现在窗口中。

第 255 行创建游戏的黑色背景。程序创建一个 `pygame Surface`，它具有和窗口一样的大小。窗口大小是通过 `screen` 的 `get_size` 方法获得的。我们希望尽可能快速地向显示区域添加对象。将表面转换成和背景一样的像素格式，就可达此目的。所以，我们对背景调用 `Surface` 的 `convert` 方法，将表面的像素格式转换成显示区域的格式。

第 258 行将背景叠加到屏幕上。第 258 行调用 `screen` 的 `blit` 方法，可在屏幕左上角位置 (0,0) 描绘背景。虽然背景已经放到 `screen` 上，但显示尚未更新。`pygame.display` 的 `update` 方法（第 259 行）重画显示区域。如果不传递任何参数，`update` 会重新显示整个 `Surface`。`update` 方法按照与 `OpenGL` 双缓冲一样的方式执行，也就是向背景缓冲区绘图，再把它切换成当前显示。

第 261~265 行载入所需的语音文件。每一行都创建一个 `Sound` 对象（在 `pygame.mixer` 中定义），它们来源于第 224~228 行创建的一个路径。第 268~276 行载入飞船图像。在这个游戏中，飞船可能有 4 种状态：左移、下移、右移和坠毁。与前三中状态的飞船图像对应的路径追加到 `shipFiles` 列表（第 231~233 行）。稍后讨论 `Player` 类时，还会对此详加解释。`for` 循环（第 270~273 行）遍历列表，载入每幅图像。第 271 行使用 `pygame.image` 的 `load` 方法载入一幅图。注意由于背景像素格式已被转换，所以载入的每幅图的像素格式也必须转换。这样可加快应用程序在显示区域放置这些图像的速度。`load` 返回的是一个 `pygame Surface` 对象，它被指派给变量 `surface`。

第 272 行调用 `surface` 的 `get_at` 方法，获得图像 (0, 0) 位置的颜色。游戏中使用的图像（比如飞船和小行星）是矩形，其中所有多余的空白都具有纯白背景色。为产生透明效果，要将 `color key` 设为不应描绘的像素的颜色。我们向 `set_colorkey` 方法传递白色。结果就是白色永远不会描绘，所以会在每个表面上呈现透明效果。每个表面都追加到 `loadedImages` 列表。第 276~277 行以类似的方式载入代表坠毁状态的图像。

第 280 行调用 `screen` 的 `get_width` 方法来获得窗口宽度。我们希望飞船出现在屏幕中央，所以把这个值的一半指派给 `centerX`。第 281 行创建一个 `Player` 对象，并把它指派给变量 `theShip`。传给 `Player` 构造函数的参数可保证飞船出现在屏幕中央，并距离顶部 25 像素。下面来讨论 `Sprite` 和 `Player` 类，它们是我们接着要插入显示区域的对象。

`Sprite` 类（第 10~28 行）定义一个 `sprite`，它是我们放在屏幕上的任何 2D 图像。`Sprite` 构造函数取得一个 `pygame Surface` 对象，名为 `image`。第 16 行将此 `Surface` 存储到类属性 `image` 中。第 17 行使用 `Surface` 的 `get_rect` 方法获得图像的“限制矩形”，并把这个矩形存储到类属性 `rectangle` 中。限制矩形定义了对象边界。`get_rect` 返回的是一个 `pygame` 的“矩形样式” (`rectstyle`) 对象。

`pygame` 的矩形样式是描述一个矩形对象的不同形式。第一种形式是含有 4 个元素的一个序列，语法格式为 `[xpos, ypos, width, height]`，其中的 `xpos` 和 `ypos` 是矩形的左上角坐标，`width` 和 `height` 规定了矩形大小。第二种形式是一对序列，表示成 `[xpos, ypos], [width, height]`。第三种形式是 `pygame.Rect` 类的一个对象。`Rect` 对象代表一个矩形。`get_rect` 返回的“矩形样式”是一个 `Rect` 对象，其中的 `xpos` 和 `ypos` 都是 0。许多 `pygame` 方法都接收“矩形样式”作为自己参数，而不是接收 `Rect` 对象（包括 `Rect` 构造函数）。对于此类方法，可能只需向方法传递由 4 个元素构成的一个序列（这也更加方便）。

接着要在屏幕上显示图像对象。`Sprite` 的 `place` 方法（第 19~22 行）在屏幕上插入一个指定对象。`place` 方法取得一个名为 `screen` 的 `Surface` 对象作为参数。第 22 行调用 `screen` 的 `blit`（叠置）方法，以便在位置 `rectangle` 处描绘对象。注意修改 `Rect` 对象 `rectangle`，就可不同位置描绘对象。接着，`place` 方法调用 `blit` 方法，并返回一个 `Rect` 对象以代表更新过的区域。

要从显示区域移除一个对象，需要使用 `Sprite` 的 `remove` 方法（第 24~28 行）在对象上描绘对象（第

27~28行)。blit方法调用取得三个参数,其中两个是Rect对象,第三个则指定要在Rect对象指定的位置描绘background的哪个区域。如果没有指定第三个参数,整个背景都会显示在由rectangle指定的位置。remove方法返回一个Rect,它代表叠置之后的区域。

然后,程序实现玩家操纵的theShip对象。Player类(第30~128行)代表由玩家控制的对象,它表面能在屏幕上运动。但事实上,游戏是通过在屏幕上移动小行星,造成飞船运动的错觉。玩家要在一个小行星带中安全行驶飞船,直到游戏结束。Player从Sprite类继承。第281行创建一个Player对象,它调用Player的构造函数(第33~43行)。第37~40行将图像表面和开始位置指派给类属性movingImages、crashImage、centerX和centerY。第41行将playerPosition设为1。变量playerPosition是所显示的图像的索引;具体显示的图像取决于飞船的方向。movingImages属性是长度为3的一个列表,其中的索引0、1和2分别代表左移、下移和右移。所以,第42行使Player开始于“下移”状态。而playerPosition属性值为-1,表明玩家目前“坠毁”。作为坠毁的一个结果,第42行会将speed属性设为0,第43行则调用loadImage方法。

按箭头键玩游戏时,loadImage方法(第45~55行)会更新Player属性。第48~51行确定要使用的图像。如果没有坠毁,程序使用代表当前玩家状态的图像(第51行)。第53行调用Sprite的构造函数来更新image和rectangle属性。第54~55行改变rectangle的centerx和centery属性,将对象移动到正确位置。

按左右箭头键,会分别调用Player的moveLeft和moveRight方法。我们把这两个相似的方法放在一起讨论。首先,if语句判断玩家是否坠毁(即playerPosition是否等于-1)。如果是,speed降为1,程序将玩家定位到障碍物左边(第62行)或右边(第73行)。如玩家不在最左或最右,那么按下左或右箭头键,程序就分别使玩家左移(第64行)或右移(第75行)一个位置。最后,loadImage方法更新图像和位置。

decreaseSpeed方法(第79~82行)行会在用户按下箭头键时调用,使speed属性减1。按下箭头键会调用increaseSpeed方法,它使speed增1(第84~92行)。第90~92行判断玩家是否坠毁。如果是,playerPosition设为1(下移),而且图像更新(第92行)。

Player的collision方法(第94~99行)会在飞船撞击一个小行星时调用。该方法将speed设为0,将playerPosition设为-1(坠毁),并调用loadImage方法。至于是否发生撞击,要通过collisionBox方法(第101~104行)返回的Rect来判断。后者调用Rect的inflate方法,并返回结果。注意我们使用较小的限制矩形来检测撞击,因为飞船图像并不完全地填充它的矩形。限制矩形相交,图像不相交,就不能判定发生了撞击,否则会使玩家非常困惑。使用较小的限制矩形进行撞击检测有时称为“子矩形撞击”(Sub-Rectangle Collision)。为获得一个较小的限制矩形,我们的做法是调用inflate方法,它返回一个新Rect。本例向方法传递的参数是-20和-20,指定要将Rect缩放多少。负参数使矩形缩小,而正参数使矩形放大。

distanceMoved方法(第114~128行)判断玩家位置变化。第119行调用isMoving方法(第106~112行),检测玩家是否正在移动。如正在移动,程序必须计算xIncrement和yIncrement。第121~126行使用playerPosition和speed来判断移动距离。如果是下移,玩家在垂直方向的移动速度是左移或右移的两倍。

Obstacle类(第130~167行)从Sprite继承。Obstacle代表玩家必须避开的一个对象,在这个游戏中就是小行星。创建一个Obstacle时,会调用它的构造函数(第133~144行)。第136行调用Sprite构造函数来初始化image和rectangle属性。

实例化一个Player后(第281行),程序将创建小行星。第284~285行载入小行星图像,并使图像的白色像素变得透明。for循环(第288~290行)创建由NUMBER_ASTEROID指定数量的小行星,每个都重复使用小行星图像。每个小行星都是Obstacle类的一个对象。传给Obstacle构造函数的参数保证每个小行星都在屏幕上随机定位。注意传给random.randrange的值大于屏幕大小,目的是保证小行星逐渐移入屏幕,而不是冒然出现。小行星的移动方向依赖于飞船的当前状态。一旦小行星离开屏幕顶部,它就再次进入屏幕底部,营造一种滚屏效果。

在这个游戏中，小行星要先完全离开屏幕（即进入负的屏幕坐标），程序才会移除它们，再放回屏幕。为达到这个目的，程序通过跟踪每个 `Obstacle` 的两个位置，从而缓冲可视区域。`rectangle` 方法代表小行星的实际位置。这是我们放置（place）对象的位置。对象属性 `positiveRectangle` 代表移入正屏幕坐标的 `rectangle` 的坐标。第 139~141 行创建并初始化 `positiveRectangle` 的位置。第 144 行调用 `Rect` 的 `move` 方法来更新 `rectangle`。在这个方法调用之后，`rectangle` 会变成和 `positiveRectangle` 一样大小的矩形，只是在 x 和 y 方向都移动 -60 像素。这样的转移使飞船离开它撞到的的小行星。

`Obstacle` 的 `move` 方法（第 146~162 行）将对象转移到新位置。`move` 方法需要获取 `xIncrement` 和 `yIncrement` 参数。记住 `Player` 类提供了 `distanceMoved` 方法。该方法返回所需的值。第 149~150 行使 `positiveRectangle` 的对象向上移动指定的量。`if` 语句（第 153 行）判断小行星是否抵达屏幕顶部。如果是，第 154~155 行使 `positiveRectangle` 的 `xpos` 与一个随机整数相加，保证小行星下次出现在屏幕上时， x 坐标不会与上次相同。这样，小行星在屏幕上就能随机出现。程序将 `Rect` 对象 `positiveRectangle` 视为含有 4 个元素的一个序列，形如 `[xpos, ypos, width, height]`。第 158~159 行确保 `positiveRectangle` 的 `xpos` 和 `ypos` 在范围之内。最后，由于 `positiveRectangle` 已更新，所以 `Rect` 的 `move` 方法（第 162 行）获得新 `rectangle` 值。

和 `Player` 类一样，`Obstacle` 的撞击检测也是用 `collisionBox` 方法返回的 `Rect` 来实现的。后者会调用 `Rect` 的 `inflate` 方法，并返回结果（第 164~169 行）。

游戏中还出现了能源块图像。玩家收集到一个能量块，会为时钟添加一定的时间。创建了小行星后（第 288~290 行），`load` 和 `convert` 方法分别载入和转换能量块图像（第 293 行），将白色设为透明色（第 294 行）。游戏期间，能量块是通过 `Objective` 类来创建的。`Objective` 类（第 169~184 行）有一个构造函数（第 172~179 行）以及一个 `move` 方法（第 181~185 行），它们类似于 `Obstacle` 类的构造函数和方法。第 296 行调用 `Sound` 的 `play` 方法来播放 `startSound`。游戏开始时就能听到这个声音。第 297 行调用 `pygame.time` 的 `set_timer` 方法，每过 1000 毫秒（1 秒）就生成一个 `USEREVENT`（用户事件）。`USEREVENT` 是 `pygame` 的一个常量，代表用户自定义的事件。第 297 行的效果是：每秒都将一个 `USEREVENT` 事件被放到事件队列中。后面会详细讨论 `pygame` 的“事件系统”。

第 299~421 行的 `while` 循环让玩家能实际地玩这个游戏。每次循环，都要判断 `courseOver` 是否为 0。如果是，表明玩家飞船尚未结束穿越小行星带，所以游戏继续。第 302~307 行使用 `pygame` 的 `time` 模块保证游戏不至于执行得太快。第 30 行调用 `pygame.time` 的 `get_ticks` 方法，它返回自从 `pygame.time` 导入以来所经历的时间（以毫秒为单位）。该值存储到变量 `currentTime` 中。如 `currentTime` 小于 `nextTime`（即上一次“滴答”数加上常量 `WAIT_TIME`），就调用 `time` 的 `delay` 方法（第 305 行），使程序暂停指定的毫秒数。传给 `delay` 的值是 `nextTime` 之前剩余的延迟毫秒数。

接着，程序更新显示。为了更新屏幕上所有对象的位置，程序移除每个对象，改变它的位置，并把它放置（或叠置）到屏幕上。然后，`pygame.display.update`（就像第 259 行那样）更新整个显示。然而，更新整个显示的效率不高，速度较慢。为加快屏幕更新，一种流行的技术是“Dirty Rectangle Animation”（变动矩形动画），即记录一系列发生变动的矩形（代表显示区域）。从屏幕移除一个对象后，那个对象的当前矩形被追加到列表，然后更新对象位置。最后，程序将对象放回屏幕，并把它的新矩形追加到列表。调用 `update` 方法时，会向其传递由“变动的”矩形构成的一个列表。结果是 `update` 只重画需要改变的部分。这样可显著提高游戏性能。注意传给 `update` 的矩形列表可以是任何“矩形样式”的一个列表。

这个游戏实现了变动矩形动画。第 310~319 行调用各自的 `remove` 方法，从屏幕移除飞船、所有小行星和能量块（如果有的话）。每个 `remove` 调用都会返回一个 `Rect`，它代表发生变化的区域。每个 `Rect` 都追加到 `dirtyRectangles` 列表。

接着讨论 `pygame` 事件处理。和 `Tkinter` 一样，事件可通过键盘或鼠标生成。`pygame` 模块还能处理其他事件，包括游戏手柄事件。`pygame` 的事件处理方法使用“事件队列”。一旦检测到事件，就把它们放到队列中。队列中的每个 `Event` 对象都有一个 `type`（类型）属性。键盘按键事件的类型是 `KEYDOWN`。大多数用户自定义事件的类型都是 `USEREVENT`。如请求退出游戏，会生成一个 `QUIT` 事件。

第 322 行调用 `pygame.event` 的 `poll` 方法，它返回在队列中等待的下一个 `Event`。该对象被指派给变

量 event。如果 event 是游戏退出请求 (QUIT)，或者是 key 属性为 K_ESCAPE 的一个 KEYDOWN 事件，程序就会退出 (第 327 行)。第 329~343 行判断 event 是否由 4 个箭头键生成 (K_UP、K_DOWN、K_RIGHT 或者 K_LEFT)。如果是，程序就调用相应的 Player 方法。记住第 297 行每过一秒就在事件队列中放入一个 USEREVENT 事件。第 346 行判断 event 是否该事件。如果是，timeLeft (穿越小行星带的剩余时间) 就会减 1 (第 347 行)。

第 350~352 行尝试新建一个能量块。如果不存在能量块 (energyPack 等于 None)，而且 randrange 返回 0，程序就从 Objective 类新建一个 energyPack。传给 Objective 构造函数的参数保证能量块出现在屏幕底部一个随机位置。假如不存在能量块，新建一个的概率是百分之一，因为方法调用向 randrange 传递 100。

接着，程序更新小行星和能量块 (如果有的话) 的位置。如飞船正在移动 (即 speed > 0)。程序从 Player 的 distanceMoved 方法获取 xIncrement 和 yIncrement (第 356 行)。程序还更新每个小行星的位置 (第 358~359 行) 和能量块的位置 (第 361~362 行)。第 364 行判断能量块是否离开屏幕顶部。如果是，就销毁当前能量块 (第 365 行)。第 367 行使 distanceTraveled 值自增。

接着，程序检测与小行星的碰撞。第 371~374 行创建名为 asteroidBoxes 的一个列表，其中包含由每个小行星的 collisionBox 方法返回的 Rect。第 377~378 行将该列表传给 Rect 的 collidelist 方法，它会返回列表中第一个重叠在基本矩形上的“矩形样式”的索引。基本矩形是从飞船的 collisionBox 方法返回的 Rect。如发现重叠，collideList 就停止检查剩余列表。如果没有发现重叠，collideList 返回 -1。如果飞船同小行星碰撞 (第 381 行)，程序将播放一个碰撞声音 (第 382 行)，并将被撞的小行星移走 (第 383 行)。第 384~385 行调用飞船的 collision 方法，并从剩余时间中减去 5 秒。

第 388~394 行判断玩家是否吃到一个能量块。第 390~391 行调用 Rect 的 colliderect 方法。colliderect 方法在目标 Rect 和“矩形样式”参数发生重叠时返回 true。吃到一个能量块后，游戏播放 chimeSound，移除能量块，并在时钟上添加 5 秒 (第 392~394 行)。

然后，程序显示时钟，便于玩家查看穿越小行星带所剩余的时间。第 397~403 行将所有对象都放回屏幕，并把它们的矩形追加到 dirtyRectangles。第 406~409 行更新屏幕左上角的时钟。updateClock 方法 (第 197~204 行) 移除以前的时钟 Surface，新建一个时钟，并把它放到屏幕。pygame.font.Font 对象 (第 199 行) 允许程序在一个 Surface 上呈现文本。Font 构造函数接收两个参数。第一个是要使用的字体文件名称。如果为 None，Font 就使用 pygame 的默认字体文件 (bluebold.ttf)。第二个参数是字号。第 199 行创建的游戏时钟使用 48 磅的 bluebold 字体。第 200~201 行调用 font 的 render 方法来创建一个新的 Surface，并在上面使用指定的字体。render 方法最多接受 4 个参数。第一个是要创建的文本。第二个指定是否使用边缘抗锯齿 (边缘平滑)。第三个参数是字体的 RGB 颜色。第四个是背景的 RGB 颜色。如果没有指定第四个参数，文本背景就是透明的。updateClock 函数返回旧的 (remove) 和新的 (post) 矩形。

时钟创建好并叠置到屏幕上之后，第 408~409 行将时钟以前的矩形和当前的矩形追加到 dirtyRectangles 中。第 412 行是矩形动画的最后一步，也就是由程序更新所有变动过的区域。没有这行代码，玩家就看不到屏幕显示的任何变化。第 413 行初始化 dirtyRectangles，为下次循环做好准备。

玩家结束穿越小行星带 (第 416 行)，程序就将 courseOver 设为 1。这可保证 while 结构在当前循环之后退出。如玩家还没有结束穿越，程序判断玩家是否超时 (第 420 行)。如果是，程序就退出 while 循环。

while 循环终止后，将从第 423 行继续执行，并判断玩家的输赢。如果是赢，游戏播放 applauseSound，并将 message 设置成“Asteroid Field Crossed!”。否则，程序播放 gameOverSound，message 设置成“Game Over!”。第 430~431 行调用 pygame.display 的 update 方法，向用户显示出 message。displayMessage 方法返回传给 update 的“矩形样式”。这个方法 (第 188~194 行) 在屏幕上放置一条消息，并返回被修改的屏幕区域。第 434~439 行的 while 循环将继续执行，直到用户退出程序。图 24.7 展示了该游戏的一个屏幕截图。

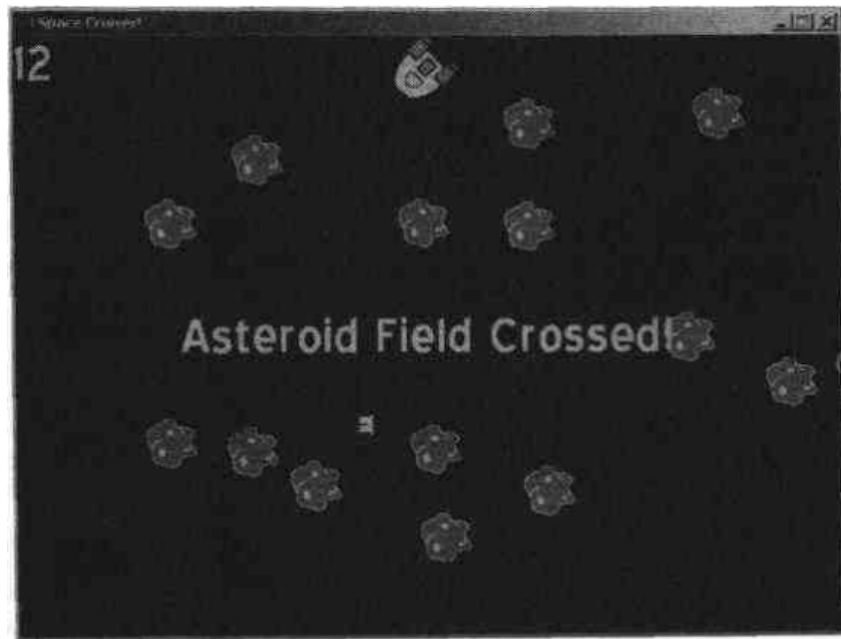


图 24.7 太空船游戏

24.10 因特网和万维网资源

pyopengl.sourceforge.net

PyOpenGL 模块主页，介绍了这个模块，并提供了文档和下载链接。

www.wag.caltech.edu/home/rpm/python_course/Lecture_7.pdf

这个教学幻灯片系列讨论了 Python 和 OpenGL 之间的交互，其中还包括几个简介性质的例子。

www.opengl.org

OpenGL 主页提供 FAQ、下载、文档和论坛。

www.alice.org

Alice 主页提供下载和文档链接。

www.pygame.org

pygame 主页提供了可供下载的 pygame 模块，还提供了文档和示范代码链接。

第 25 章 Python 服务器页 (PSP)

学习目标

- 会创建和部署 PSP
- 会用 Python 和 PSP 的隐式对象来创建动态网页
- 理解 PSP 动作
- 会在 XHTML 中嵌入 Python Scriptlet
- 会用预编译指令指定全局 PSP 信息
- 创建一个基于 XML 的论坛程序

25.1 概述

本章介绍 Python 服务器页 (PSP)，它简化了动态 Web 内容的生成与递送。利用 PSP，Web 应用程序开发者可重用预定义组件，并通过服务器端脚本与组件交互，从而创建动态内容。

本章使用了 Webware，它是帮助开发者用 Python 生成动态 Web 内容的一个套装软件。Webware 的 WebKit 包相当于一个“Python Servlet 引擎”，它可生成 XHTML。WebKit 是一个应用程序服务器，负责管理及执行 Python Servlet，后者是响应客户请求并在 Web 服务器上执行的一种 Python 对象。Webware 的 PSP 能力是基于 WebKit 而构建的，所以事先必须安装和配置好 WebKit 包。具体的安装和配置指击请参见 www.deitel.com。本章的例子使用 Webware 0.6 和 Python 2.2b2。另外，本章介绍的 Python Servlet 和 PSP 是 Webware 所特有的，后者是来自 Python Software Foundation 的一个独立实体。

WebKit 提供一系列核心类来创建 Python Servlet 和 PSP。另外还有一些辅助类，用于提供 Web 开发者经常需要的功能。用于 Servlet 编程的类存储在 Webware/WebKit 目录；PSP 编程所需的类存储在 Webware/PSP 目录。本章在讲解 PSP 的基础知识时，会顺便讨论许多这样的类。要想了解 PSP 更多的信息，请访问 Webware/PSP/Docs/UsersGuide.htm。

25.2 Python Servlet^①

讨论 PSP 之前，先来简单认识一下 Python Servlet。客户向 Web 服务器发送一个 PSP 请求后，Web 服务器会调用“Servlet 容器”，也就是管理 Servlet 请求的一个进程。在 Webware 中，AppServer 就是 Servlet 容器。一旦调用，容器就会将 PSP 转换成一个新程序，即一个“Servlet”，由它来处理 PSP 请求。

Servlet 的生命期开始于 Servlet 容器将 Servlet 载入内存的时候（通常是为了响应 Servlet 接收到第一个请求）。Servlet 为了处理一个请求，必须先由 Servlet 容器调用 Servlet 的 `awake` 方法来初始化 Servlet。awake 完成后，Servlet 才能响应请求。Servlet 的 `respond` 方法接收并处理请求，并向客户发送一个响应。请求完成后，容器调用 Servlet 的 `sleep` 方法来释放 Servlet 资源。默认情况下，Servlet 只需构造一次，即可处理许多请求。在 Servlet 的生命期中，容器为 Servlet 处理的每个请求都调用 `awake`、`respond` 和 `sleep` 方法。

Servlet 容器使用由 WebKit 提供的类和方法。客户向 Web 服务器发送一个 PSP 请求后，WebKit 的 ServletFactory（一个抽象类，定义了应用程序创建 Servlet 所需的协议）会创建一个 Servlet 实例来处理请求。Web 应用程序使用 ServletFactory 为事务处理创建相应的 Servlet。在 Webware 中，每个“事务处理”都是一个对象，包含一个请求 - 响应循环所牵涉到的所有对象。客户的 Web 浏览器每次取得一个页

^① Python Servlet 的细节超出了本章范围。不必掌握 Python Servlet 就能理解本章内容。有兴趣深入学习 Python Servlet 的读者可访问：webware.sourceforge.net/Webware/WebKit/Docs/Source/ClassHier.html。

时，都会新建一个事务处理对象，并在结束该页的处理之后销毁。在事务处理中牵涉到的类包括 Application、Request、Response、Session 和 Servlet。

Application 类和 WebKit 中的各种类一起，管理（即创建和销毁）事务处理中的 Servlet。Request 类包含和一个客户的请求有关的信息（例如发出请求的时间和位置）。Web 开发者通常使用 HTTPRequest 子类，它添加了会话跟踪、Cookie 和身份验证等功能，以扩展 Request 类的能力。Response 类负责向客户发送一个响应。类似于 HTTPRequest，Web 开发者经常通过 HTTPResponse 子类扩展 Response 类的能力（比如 Cookie 支持），以简化响应头信息的处理。Session 类创建一个会话对象，它在指定的时间内同特定用户关联在一起；PSP 使用 Session 对象在服务器上跟踪用户信息（例如用户访问过的页）。

Servlet 的 HTTPServlet 子类支持 get、post、put、delete、options 和 trace 请求。get 请求从服务器获得数据，post 向服务器发送数据。put 或 delete 请求分别在服务器上存储或删除文件。options 请求向客户返回信息，指出服务器支持的 HTTP 选项。trace 请求通常用于调试。Page 类是 HTTPServlet 的一个子类，提供了一些便利的方法来创建 XHTML 页，以响应 get 和 post 请求。开发 Servlet 时，Web 开发者经常要使用 Page 的子类，以继承它的实现，并覆盖它的方法。图 25.1 展示了 Servlet 类支持的方法。

方法	说明
name()	返回类名
awake(transaction)	由 Servlet 程序员定义，用于初始化 Servlet。transaction 参数由执行 Servlet 的 Servlet 容器提供
respond(transaction)	Servlet 容器调用该方法来响应客户的 Servlet 请求
sleep(transaction)	Servlet 由其容器终止时，会调用这个“清除”方法。Servlet 使用的资源（比如打开的文件或者临时数据库连接）应在此回收。不再需要的 Servlet 属性应该删除
log(message)	这个方法打印 Servlet 信息，它们有助于调试
canBeThreaded()	如果 Servlet 可以多线程处理，该方法返回 1；否则返回 0。默认返回 0
canBeReused()	如果 Servlet 可以重用，该方法返回 1；否则返回 0。默认返回 1
serverSidePath(path)	该方法返回页在服务器上的路径

图 25.1 Servlet 类的方法 (WebKit/Servlet.py)

25.3 PSP 简介

PSP 有三个关键组件：预编译指令、动作和 Scriptlet。预编译指令是发送给 PSP 容器的消息，以便由程序员指定页面设置，以及包括来自其他资源的内容。Scriptlet 是脚本编程元素，允许程序员插入 Python 代码，以便与 PSP 中的组件（以及其他 Web 应用程序组件）交互，从而执行请求处理。动作将功能封装在预定义标记中，以便程序员将其嵌入 PSP。动作通常基于发送给服务器的、作为特定客户请求一部分的信息。另外，动作可创建 Python 对象以便在 PSP Scriptlet 中使用。这些 PSP 组件的详情将在以后的小节讨论。

PSP 在许多方面类似于标准 HTML 或 XHTML 文档。事实上，PSP 通常包括 XHTML 或 HTML 标记。这种标记称为“固定模板数据”或“固定模板文本”。对固定模板数据的需要可帮助程序员决定使用 Servlet 还是 PSP。如果发送给客户的大多数内容都是固定模板数据，只有一小部分内容使用 Python 代码动态生成，程序员应使用 PSP。如果只有一小部分发送给客户的内容是固定模板数据，程序员应使用 Servlet。事实上，有的 Servlet 不会生成内容。相反，它们代表客户执行任务，再调用其他 Servlet 或 PSP 来提供响应。注意大多数情况下，Servlet 和 PSP 技术是可互换的。和 Servlet 一样，PSP 通常作为 Web 服务器的一部分执行。

支持 PSP 的服务器收到第一个 PSP 请求后，PSP 容器将该请求转换成一个 Python Servlet，以处理当前和未来的 PSP 请求。编译新 Servlet 的过程中出错，会导致“生成时错误”。PSP 可直接响应请求，或者调用其他 Web 应用程序组件以帮助处理请求。请求处理期间发生的任何错误都称为“请求时错误”。

总之, PSP 的请求/响应机制和生命期和 Servlet 相同。PSP 可覆盖来自 Servlet 类的 `awake` 和 `sleep` 方法。初始化和终止 PSP 时, PSP 容器会分别调用这两个方法。PSP 程序员可利用 PSP 页的预编译指令和动作来定义这些方法。

25.4 第一个 PSP 示例

下面用一个简单的例子开始 PSP 的学习 (图 25.2), 它将当前日期和时间插入网页。程序使用 `page` 预编译指令导入 `time` 模块, 并在 PSP 表达式中使用模块的 `ctime` 函数插入日期和时间。

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4 <!-- Fig. 25.2: fig25_02.psp -->
5 <!-- Displays date and time every 60 seconds. -->
6
7 <%-- import time module --%>
8 <%@ page imports = "time" %>
9
10 <html xmlns = "http://www.w3.org/1999/xhtml">
11
12   <head>
13     <meta http-equiv = "refresh" content = "60" />
14
15     <title>A Simple PSP Example</title>
16
17     <style type = "text/css">
18       .big { font-family: helvetica, arial, sans-serif;
19             font-weight: bold;
20             font-size: 2em; }
21     </style>
22   </head>
23
24   <body>
25     <p class = "big">Simple PSP Example</p>
26
27     <table style = "border: 6px outset;">
28       <tr>
29         <td style = "background-color: black;">
30           <p class = "big" style = "color: cyan;">
31
32             <!-- PSP expression to insert date/time -->
33             <%= time.ctime() %>
34
35           </p>
36         </td>
37       </tr>
38     </table>
39   </body>
40
41 </html>

```

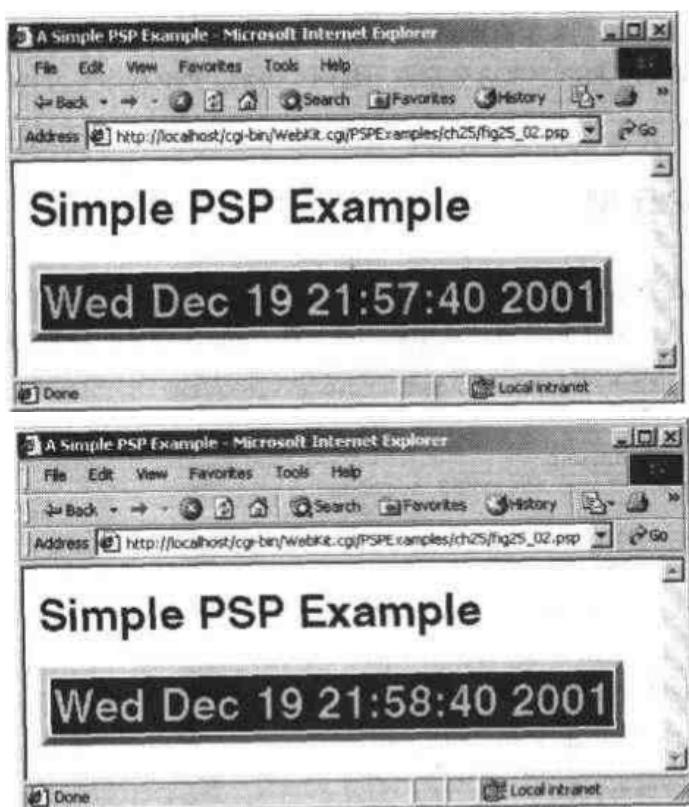


图 25.2 PSP 的 page 预编译指令和动态插入日期/时间的 Python 表达式

程序中的大多数代码都由 XHTML 标记构成。拿图 25.2 的情况来说, PSP 比 Servlet 更容易实现。如果用 Servlet 来执行和这个 PSP 相同的任务, 每一行 XHTML 标记通常都是单独的 Python 语句, 以输出代表标记的字符串。为输出标记编写代码容易出错。

软件工程知识 25.1 如果响应主要由多次请求都保持不变的标记构成, PSP 比 Servlet 更易实现。

图 25.2 的 PSP 生成一个能显示当前日期和时间的 XHTML 文档。这个 PSP 最关键的一行(第 33 行)是以下表达式:

```
<%= time.ctime() %>
```

PSP 表达式用 `<%=` 和 `>` 定界。上述表达式调用 `time` 模块的 `ctime` 函数。客户请求 PSP 时, 上述表达式将代表日期和时间的字符串插入响应中。为使用 `time` 模块, PSP 首先必须将其导入。第 8 行通过设置 page 预编译指令的 `imports` 属性, 从而导入该模块。page 预编译指令指定了当前 PSP 的设置。

`imports` 属性指示 PSP 解析器导入一个要在此页使用的 Python 模块。25.8.1 节将详细讲解 page 预编译指令。

软件工程知识 25.2 PSP 容器将每个 PSP 表达式的结果转换成一个字符串, 以便作为响应的一部分输出。

注意我们在第 13 行使用 XHTML meta 元素为文档设置 60 秒的“刷新周期”。这会导致浏览器每过 60 秒便请求一次 `fig25_02.psp`。对于每个请求, PSP 容器都重新求值第 33 行的表达式, 从而获得服务器的当前日期和时间。

我们使用 Webware 的应用程序服务器 (AppServer) 和 Apache Web 服务器来测试我们的 PSP。要测试 `fig25_02.psp`, 请在 `Webware/PSP/Examples` 目录中新建一个 `ch25` 子目录。然后将 `fig25_02.psp` 拷贝到这个子目录, 并启动 Apache。打开一个命令提示符窗口, 转到 `Webware/WebKit` 目录, 并输入 AppServer (对于 UNIX 和 Linux 用户是 `/AppServer`), 从而启动应用程序服务器。打开 Web 浏览器, 输入以下 URL

来测试 fig25_02.psp:

`http://localhost/cgi-bin/WebKit.cgi/PSPEXamples/ch25/fig25_02.psp`

首次调用 PSP 时, WebKit 会将 PSP 转换成一个 Servlet, 再调用该 Servlet 来响应请求。注意并非必须在 Webware 中创建 ch25 目录。用这个目录只是为了将本章的例子和 Webware 提供的其他例子区分开。

25.5 隐式对象

“隐式对象”(Implicit Object)使程序员能在 PSP 环境中使用许多 Servlet 功能。图 25.3 总结了这些 PSP 隐式对象, 它们相当于 PSP 实例的局部变量。所有 PSP 都能访问这些对象。

注意隐式对象扩展了 25.2 节讨论的类。所以, PSP 可使用与 Servlet 相同的方法和这些对象交互。本章的大多数例子都使用了图 25.3 列出的一个或多个隐式对象。

隐式对象	说明
trans	在 Webware 中, 事务处理是一个对象, 其中包含单个请求 - 响应周期牵涉到的全部对象。这个 WebKit.Transaction 对象代表在其中执行 PSP 的容器
res	该对象代表对客户的响应, 是 WebKit.HTTPResponse 类的一个实例
req	该对象代表客户请求, 是 WebKit.HTTPRequest 类的一个实例

图 25.3 PSP 隐式对象

25.6 脚本编程

PSP 通常将动态生成的内容作为响应给客户的每个 XHTML 文档的一部分。内容有时是静态的, 只在满足特定条件时才会输出(比如在表单中输入值并提交一个请求)。PSP 程序员可通过脚本编程在 PSP 中插入 Python 代码和逻辑。

25.6.1 脚本编程组件

PSP 脚本编程组件包括 Scriptlet、注释和表达式。本节对其进行了描述。许多组件都在 25.6.2 节最后的图 25.4 中进行了演示。

Scriptlet 是用 `<%和%>` 定界的代码块。PSP 支持三种注释样式: PSP 注释、XHTML 注释和来自脚本语言的注释。PSP 注释用 `<%--和--%>` 定界。这种注释可放在 PSP 文档的任何地方, 只是不能放在 Scriptlet 内。XHTML 注释用 `<!--和-->` 定界, 允许的位置和 PSP 注释相同。Scriptlet 则可使用 Python 的单行注释(用 `#` 定界)和多行文档字符串(用 `"""` 和 `"""` 定界)。

常见编程错误 25.1 将 PSP 或 XHTML 注释放到 Scriptlet 内会造成生成时语法错误, PSP 不能正确转换。

PSP 和 Python 注释会被忽略, 不会出现在对客户的响应中。客户查看 PSP 响应文档的源程序时, 只能看到 XHTML 注释。不同的注释样式有助于区分应该让用户看到的注释和服务端上的程序逻辑文档注释。

PSP 表达式用 `<%=和%>` 定界, 包含一个标准的 Python 表达式, 并在客户请求包含它的 PSP 时进行求值。容器将 PSP 表达式结果转换成一个字符串, 再把它作为响应的一部分输出到客户。

25.6.2 脚本编程示例

图 25.4 的 PSP 响应 get 请求来演示基本的脚本编程能力。这个 PSP 允许用户输入自己的名字，再把那个名字作为响应的一部分输出。通过脚本编程，PSP 判断一个 firstName 参数是否作为请求的一部分传给 PSP；如果不是，PSP 就返回一个 XHTML 文档，用户可利用其中的表单输入名字。否则，PSP 获得 firstName 值，并在 XHTML 文档中把它用在一条欢迎辞中。

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4  <!-- Fig. 25.4: fig25_04.psp -->
5  <!-- PSP that processes a "get" request containing data. -->
6
7  <!-- specify indent type --%>
8  <%$ page indentType = "braces" %>
9
10 <html xmlns = "http://www.w3.org/1999/xhtml">
11
12   <!-- head section of document -->
13   <head>
14     <title>Processing "get" requests with data</title>
15   </head>
16
17   <!-- body section of document -->
18   <body>
19
20     <!-- generate a form --%>
21     <% # begin scriptlet
22
23       if req.hasField( "firstName" ): {
24
25     %> <!-- end scriptlet to insert fixed template data --%>
26
27     <h1>
28       Hello <%= req.field( "firstName" ) %>, <br />
29       Welcome to Python Server Pages!
30     </h1>
31
32     <% # continue scriptlet
33
34     } # end if
35     else: {
36
37     %> <!-- end scriptlet to insert fixed template data --%>
38
39     <form action = "fig25_04.psp" method = "get">
40       <p>Type your first name and press Submit</p>
41
42       <p><input type = "text" name = "firstName" />
43         <input type = "submit" value = "Submit" />
44       </p>
45     </form>
46
47     <% # continue scriptlet
48
49     } # end else
50
51     %> <!-- end scriptlet --%>
52   </body>
53 </html> <!-- end XHTML document -->

```



图 25.4 PSP 脚本编程示例 (fig25_04.psp)

类似于图 25.2 的程序, 图 25.4 的大多数代码都是 XHTML 标记 (即固定模板数据)。body 元素内包含几个 Scriptlet (第 21~25 行、第 32~37 行和第 47~51 行) 以及一个 PSP 表达式 (第 28 行)。注意 3 种注释样式都在这个 PSP 中出现。

对于每个请求, 第 1~6 行和第 9~19 行的固定模板数据将作为响应的一部分发送给客户。第 21 行开始一个 Scriptlet, 其中含有一个 if 求值语句 (第 23 行), 它使用 PSP 隐式对象 req (一个 HTTPRequest 对象) 的 hasField 方法来判断 PSP 是否有一个名为 firstName 的参数。注意第 23 和第 34 行用花括号 ({}) 来定界一个 if suite 的代码。记住 Python 代码使用空白字符来标识代码 suite 和 block。但这种缩进类型妨碍了 PSP 的可读性。为创建更易读的 XHTML 代码, Webware 使用 3 种特殊语法来处理 Python 代码块。本例使用的是其中之一, 即花括号, 这是在第 8 行指定的:

```
<%@ page indentType = "braces" %>
```

它向编译器指出当前 PSP 使用花括号标识 Scriptlet 中的代码 suite 和 block。花括号要出现在 suite 或 block 的开头和结尾。在花括号内部, 每行开头的制表符和其他空白字符会被忽略。25.8.1 节详细讨论了 page 预编译指令 indentType。

软件工程知识 25.3 page 预编译指令一旦将花括号设为缩进类型, 必须为所有 suite 和 block 使用花括号。

如果第 23 行的条件求值为 true, 第 21~25 行就临时终止, 以便输出第 27~30 行的固定模板数据。第 28 行的 PSP 表达式输出作为一个请求参数传给 PSP 的名字。Scriptlet 在第 32~37 行继续, 其中包含 if 结构主体的结束花括号, 以及 if/else 结构的 else 部分的开头。如果第 23 行的条件是 false, 第 27~30 行不会输出; 相反, 第 39~45 行会输出一个 form 元素, 允许用户在对 fig24_04.psp 的一个 get 请求中提交名字。用户可在表单中输入名字, 再按 Submit 按钮再次请求 PSP, 并执行 if 结构主体 (第 27~30 行)。

软件工程知识 25.4 可在 PSP 中混用 Scriptlet、表达式和固定模板数据, 为每个 PSP 请求都创建相应的动态响应。

测试和调试提示 25.1 有时很难调试 PSP 中的错误, 因为 PSP 容器所报告的行号针对的是代表转换过的 PSP 的 Servlet, 而不是原始的 PSP 行号。在 Servlet 中查找报告的行号, 有助于调试原始 PSP。Servlet 位于 Webware/WebKit/Cache/PSP 目录。

为了在 Webware 中测试图 25.4 的程序, 请将 fig25_04.psp 拷贝到 Webware/PSP/Examples/ch25 目录。

确定正在运行 Webware 的应用程序服务器 (AppServer) 和 Apache Web Server。打开 Web 浏览器, 输入以下 URL 来测试 fig25_04.psp:

```
http://localhost/cgi-bin/WebKit.cgi/PSPEXamples/ch25/fig25_04.psp
```

首次执行 PSP, 它会显示表单以便输入名字。提交后, 浏览器会出现和图 25.4 的第二幅屏幕截图相似的显示。可将 get 请求参数作为 URL 的一部分传递。以下 URL 直接向 fig25_04.psp 提供 firstName 参数:

```
http://localhost/cgi-bin/WebKit.cgi/PSPEXamples/ch25/fig25_04.psp?firstName=Paul
```

25.7 标准动作

我们将通过 PSP 标准动作 (如图 25.5 所示) 继续讨论 PSP。通过这些动作, PSP 程序员可以访问在 PSP 中执行的几个最常用任务, 如包容来自其他资源的内容, 创建或覆盖一个 Servlet 类的方法。PSP 容器在请求时处理动作。动作由 `<psp:action>` 和 `</psp:action>` 标记定界, 其中的 *action* 是标准动作名, 也就是 `include`、`insert` 或者 `method`。只有 `method` 动作才必须使用结束标记 `</psp:action>`; 对于其他所有动作, 结束标记都是可选的。下面几个小节将使用动作标记。

动作	说明
<code><psp:include></code>	将另一个资源动态包容到 PSP 中。PSP 执行时, 引用的资源将被包容和处理
<code><psp:insert></code>	在 PSP 中插入另一个资源。PSP 执行时, 引用的资源会被包容, 但 PSP 内容不进行解析
<code><psp:method></code>	为 PSP 生成的 Servlet 类声明新方法, 或者覆盖基类方法 (PSP 从这个基类扩展)。必须用 <code></psp:method></code> 指定方法结束

图 25.5 PSP 标准动作

25.7.1 `<psp:include>` 动作

PSP 支持两种包容机制——`<psp:include>` 动作和 `include` 预编译指令。`<psp:include>` 动作允许 PSP 重用来自现有 PSP、XHTML 文档等等的内容。在不同请求之间, 如果包容的资源发生改变, 以后请求含有 `<psp:include>` 动作的 PSP 时, 会反映出对包容资源进行的那些改变。另一方面, `include` 预编译指令在 PSP 生成时, 将内容一次性拷贝到 PSP 中。如包容的资源发生改变, 新内容不会在使用 `include` 预编译指令的 PSP 中反映出来, 除非 WebKit 应用程序服务器重启。图 25.6 描述了 `<psp:include>` 动作的属性。

属性	说明
path	指定包容资源的 URL 路径。PSP 解析器支持绝对和相对路径。相对路径不以 <code>"/</code> 开头, 而且相对于当前页。绝对路径以 <code>"/</code> 开头, 向 PSP 解析器指出这是一个绝对路径, 而不是相对于当前页的一个路径。应用程序服务器必须能够定位该页

图 25.6 `<psp:include>` 动作属性

性能提示 25.1 `<psp:include>` 动作比 `include` 预编译指令灵活, 但在页面内容频繁变化时, 开销也更大。只有在必须使用动态内容时, 才应使用 `<psp:include>` 动作。

常见编程错误 25.2 应用程序服务器找不到 `<psp:include>` 动作中指定的页, 会造成请求时错误。

下例使用代表静态和动态内容的 XHTML 和 PSP 文档来演示 `<psp:include>` 动作。banner.html (图 25.7) 和 toc.html (图 25.8) 是静态 XHTML 文档; clock2.psp (图 25.9) 是 PSP。include.psp (图 25.10) 合并这 3 个文档以新建一个 XHTML 文档。banner.html 在表格顶部占据两列空间, toc.html 是第二行的左列,

而 clock2.psp (图 25.2 的一个简化版本) 是第二行的右列。图 25.10 使用三个<psp:include>动作 (第 38 行、第 47 行和第 54 行) 作为表格 td 元素中的内容。图 25.10 演示了同时包容动态和静态内容的 PSP。

```

1 <!-- Fig. 25.7: banner.html -->
2 <!-- Banner to include in another document. -->
3
4 <div style = "width: 500px">
5     <p>
6         Java(TM), C, C++, Visual Basic(R),
7         Object Technology, and <br />Internet and
8         World Wide Web Programming Training&nbsp;<br />
9         On-Site Seminars Delivered Worldwide
10    </p>
11
12    <p>
13        <a href = "mailto:deitel@deitel.com">
14            deitel@deitel.com</a><br />
15
16        978.579.9911<br />
17        490B Boston Post Road, Suite 200,
18        Sudbury, MA 01776
19    </p>
20 </div>

```

图 25.7 要添加到图 25.10 创建的 XHTML 文档顶部的 banner.html

```

1 <!-- Fig. 25.8: toc.html -->
2 <!-- Contents to include in another document. -->
3
4 <p><a href = "http://www.deitel.com/books/index.html">
5     Publications/BookStore
6 </a></p>
7
8 <p><a href = "http://www.deitel.com/whatsnew.html">
9     What's New
10 </a></p>
11
12 <p><a href = "http://www.deitel.com/books/downloads.html">
13     Downloads/Resources
14 </a></p>
15
16 <p><a href = "http://www.deitel.com/faq/index.html">
17     FAQ (Frequently Asked Questions)
18 </a></p>
19
20 <p><a href = "http://www.deitel.com/intro.html">
21     Who we are
22 </a></p>
23
24 <p><a href = "http://www.deitel.com/index.html">
25     Home Page
26 </a></p>
27
28 <p>Send questions or comments about this site to
29     <a href = "mailto:deitel@deitel.com">
30         deitel@deitel.com
31     </a><br />
32     Copyright 1995-2002 by Deitel & Associates, Inc.
33     All Rights Reserved.
34 </p>

```

图 25.8 要包容到图 25.10 创建的 XHTML 文档左侧的 toc.html

```

1 <!-- Fig. 25.9: clock2.psp -->
2 <!-- Date and time to include in another document. -->
3
4 <%-- import time module --%>
5 <%@ page imports = "time"%>
6

```

```

7 <table>
8   <tr>
9     <td style = "background-color: black">
10      <p class = "big" style = "color: cyan; font-size: 3em;
11        font-weight: bold">
12
13        <%= time.ctime() %>
14      </p>
15    </td>
16  </tr>
17 </table>

```

图 25.9 clock2.psp 动态生成由 include.psp 创建的 XHTML 文档的主要内容

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4 <!-- Fig. 25.10: include.psp -->
5 <!-- PSP that includes both static and dynamic content. -->
6
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8
9   <head>
10     <title>Using psp:include</title>
11
12     <style type = "text/css">
13       body {
14         font-family: tahoma, helvetica, arial, sans-serif;
15       }
16
17       table, tr, td {
18         font-size: .9em;
19         border: 3px groove;
20         padding: 5px;
21         background-color: #dddddd;
22       }
23     </style>
24   </head>
25
26   <body>
27     <table>
28       <tr>
29         <td style = "width: 160px; text-align: center">
30           <img src = "images/logotiny.png"
31             width = "140" height = "93"
32             alt = "Deitel & Associates, Inc. Logo" />
33         </td>
34
35         <td>
36
37           <!-- include contents of banner.html here -->
38           <psp:include path = "banner.html" >
39
40           </td>
41       </tr>
42
43       <tr>
44         <td style = "width: 160px">
45
46           <!-- include contents of toc.html here -->
47           <psp:include path = "toc.html" >
48
49           </td>
50
51         <td style = "vertical-align: top">
52
53           <!-- include clock2.psp in this PSP -->
54           <psp:include path = "clock2.psp" >
55
56           </td>

```



```

57         </tr>
58     </table>
59 </body>
60 </html>

```

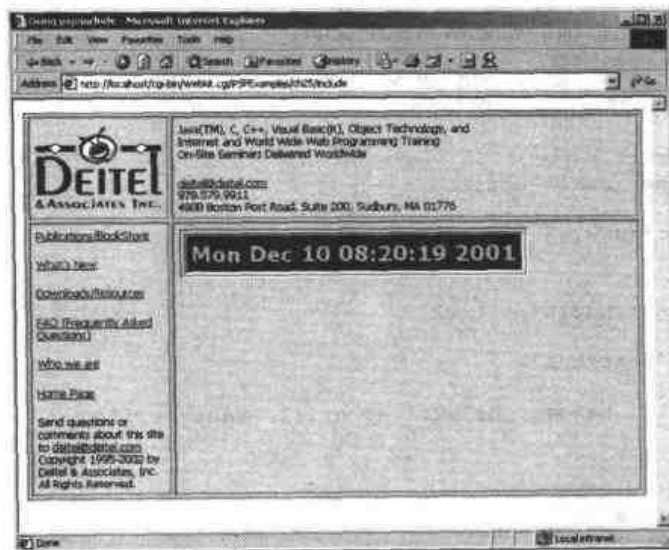


图 25.10 include.psp 使用<psp:include>添加资源

为测试图 25.10, Webware 的应用程序服务器 (AppServer) 和 Apache Web 服务器必须运行。将 banner.html、toc.html、clock2.psp、include.psp 和 images 目录拷贝到 PSP/Examples/ch25 目录。打开 Web 浏览器, 输入以下 URL 来测试 include.psp:

http://localhost/cgi-bin/WebKit.cgi/PSPEXamples/ch25/include.psp

注意运行 PSP 时, 我们提供的是完整文件名 (例如 include.psp)。但是, 即使 URL 中没有指定文件扩展名, WebKit 也允许运行一个 PSP、Servlet 或者静态 XHTML 页。利用这个特点, 网站管理员可将一个页从 PSP 变成 Servlet, 甚至变成 XHTML 页, 同时无需更改到那个页的所有链接。例如第 54 行:

```
<psp:include path = "clock2.psp" >
```

可替换成:

```
<psp:include path = "clock2" >
```

25.7.2 <psp:insert>动作

<psp:insert>动作具有和<psp:include>动作相同的功能, 也能在 PSP 中动态插入资源。但和<psp:include>不同的是, <psp:insert>指示 PSP 解析器不要对包含在资源中的任何 Scriptlet 进行解析。图 25.11 总结了<psp:insert>动作的属性。

属性	说明
file	指定要插入的资源的文件名。PSP 解析器支持绝对和相对路径。例如 PSP 在 ch25 目录中执行时, "test.psp" 等价于 "/Webware/PSP/Examples/ch25/test.psp"
static	这是可选属性。如设为 "true" 或者 "1", 插入文件的内容会在 PSP 生成时一次性拷贝到 PSP。如果插入的资源发生改变, 新内容不会在 PSP 中反映

图 25.11 <psp:insert>动作属性

下例（图 25.12）演示了<psp:include>和<psp:insert>动作的差异。类似图 25.10，图 25.12 使用三个<psp:insert>动作（第 38 行、第 47 行和第 54 行）作为表格的 td 元素的内容。所有插入动作都用相对路径指定要包容的文件，这些文件位于和图 25.12 的 PSP 相同的目录。第 38 行将 static 属性设为"true"，它指出对 banner.html 的更新不在后续的服务器响应中反映。

```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4  <!-- Fig. 25.12: insert.psp      -->
5  <!-- Demonstrates the insert action. -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8
9    <head>
10      <title>Using psp:insert</title>
11
12      <style type = "text/css">
13        body {
14          font-family: tahoma, helvetica, arial, sans-serif;
15        }
16
17        table, tr, td {
18          font-size: .9em;
19          border: 3px groove;
20          padding: 5px;
21          background-color: #dddddd;
22        }
23      </style>
24    </head>
25
26    <body>
27      <table>
28        <tr>
29          <td style = "width: 160px; text-align: center">
30            <img src = "images/logotiny.png"
31              width = "140" height = "93"
32              alt = "Deitel & Associates, Inc. Logo" />
33          </td>
34
35          <td>
36
37            <!-- include banner.html in this PSP --%>
38            <psp:insert file = "banner.html" static = "true" >
39
40          </td>
41        </tr>
42
43        <tr>
44          <td style = "width: 160px">
45
46            <!-- include toc.html in this PSP --%>
47            <psp:insert file = "toc.html" >
48
49          </td>
50
51          <td style = "vertical-align: top">
52
53            <!-- include clock2.psp in this PSP --%>
54            <psp:insert file = "clock2.psp" >
55
56          </td>
57        </tr>
58      </table>
59    </body>
60  </html>

```

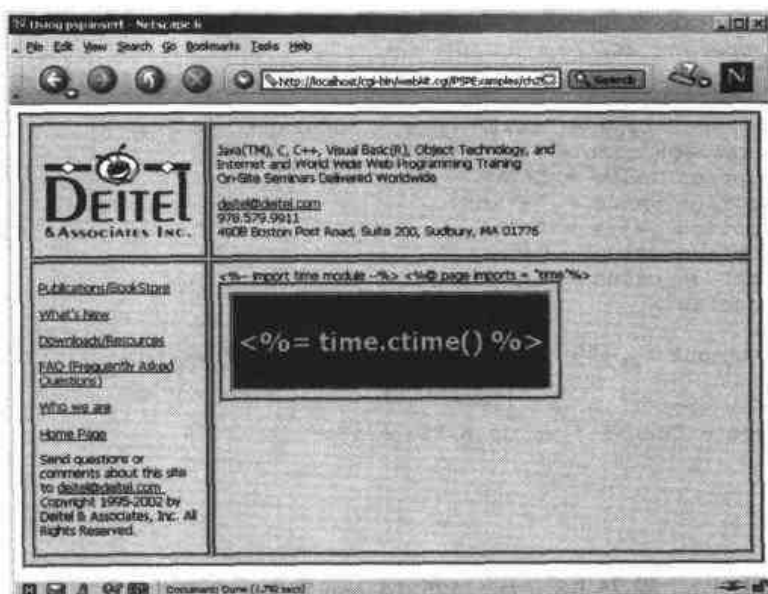


图 25.12 insert.psp 用<psp:insert>包容资源

25.12 的输出表明, <psp:insert>动作在插入一个文档时, 没有对文档中的任何 PSP 脚本编程元素进行解释。<psp:insert>动作要么在运行时动态插入文档内容, 要么在页面生成时一次性插入。具体则取决于状态标志的值 (true 表明在页面生成时一次性插入)。在示范文档右下角的单元格中, 注意出现的是原始的 PSP 代码, 而不是脚本解析的结果。

性能提示 25.2 如果<psp:insert>动作的 static 属性设为"true", 要比不设为"true"时有效, 因为每次请求文档时, 不会对插入动作进行解释。但是, 假如文档改变, 程序员必须重新启动 WebKit 应用程序服务器 (AppServer)。

25.7.3 <psp:method>动作

<psp:method>动作允许 PSP 覆盖从 Servlet 类继承的方法, 并可声明新方法。<psp:method>动作有两个属性, 即 name 和 params, 分别指定方法名和方法的参数名。位于 psp:method 元素之间的代码是标准的 Python 代码。原生 XHTML 或脚本都不允许在 psp:method 元素中出现。

fig25_13.html 中的 XHTML 创建一个表单, 提示用户输入两个数字, 并在 5 种运算中选择——加、减、乘、True 除法或 Floor 除法。单击 Submit 按钮, 就会调用 fig25_14.psp。

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4 <!-- Fig. 25.13: fig25_13.html -->
5 <!-- Calculator form. -->
6
7 <html xmlns = "http://www.w3.org/1999/xhtml">
8
9   <head>
10     <title>Calculator Input Data</title>
11   </head>
12
13   <body>
14     <h1>Python Calculator</h1>
15
16     <form method = "post" action = "fig25_14.psp">
17       <table width = "100%" border = "3">
18         <tr>
19           <th>operand 1</th>

```

```

20         <th>Operator</th>
21         <th>operand 2</th>
22     </tr>
23     <tr>
24         <td><input type = "text" name = "operand1"/></td>
25         <td><select name = "operator">
26             <option value = "+">+
27             <option value = "-">-
28             <option value = "*">*
29             <option value = "/">/
30             <option value = "//">//
31         </select>
32     </td>
33     <td><input type = "text" name = "operand2"/></td>
34 </tr>
35 </table><br />
36 <input type = "submit" value = "Submit">
37 </form>
38
39 </body>
40 </html>

```

图 25.13 为 fig25_14.psp 定义一个表单

图 25.14 演示了如何在 PSP 中声明和调用一个方法。第 8~10 行是 page 预编译指令，它指定了当前 PSP 的页面设置。第 8 行将缩进样式设为“spaces”，第 9 行将缩进空格数设为 3。第 10 行的 page 预编译指令启用 True 除法（参见第 2 章）。

第 13~36 行声明 calculate 方法，其中只包含 Python 代码。第 13~14 行包含方法的起始标记：

```

<psp:method name = "calculate"
    params = "operand1, operator, operand2" >

```

calculate 方法取得三个参数：operand1、operator 和 operand2。第 16~32 行使用 operator 对 operand1 和 operand2 执行计算。第 34 行返回计算结果。</psp:method>结束标记终止方法（第 36 行）。

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3
4 <!-- Fig. 25.14: fig25_14.psp -->
5 <!-- PSP that processes a "get" request containing data. -->
6
7 <!-- specify indent type -->
8 <%@ page indentType = "spaces" %>
9 <%@ page indentSpaces = "3" %>
10 <%@ page imports = "__future__ :division" %>
11
12 <!-- define a psp method -->
13 <psp:method name = "calculate"
14     params = "operand1, operator, operand2" >
15
16 if operator == "+":      # addition operation
17     result = operand1 + operand2
18
19 elif operator == "-":    # subtraction operation
20     result = operand1 - operand2
21
22 elif operator == "*":    # multiplication operation
23     result = operand1 * operand2
24
25 elif operator == "/":    # division operation
26     result = operand1 / operand2
27
28 elif operator == "//":   # floor division
29     result = operand1 // operand2
30
31 else:                    # unrecognizable operator
32     result = None
33

```